

21世纪高等学校计算机规划教材

21st Century University Planned Textbooks of Computer Science

C++面向对象程序设计

C++ Object-Oriented Programming

陈维兴 陈昕 编著

- 不追求深奥的理论，强调C++的实践技能
- 不追求抽象的概念，注重C++的实际应用
- 不追求完整的语法，突出C++的基本内容



精品系列



人民邮电出版社

POSTS & TELECOM PRESS

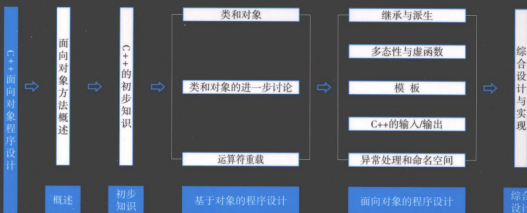
C++ Object-Oriented Programming

C++面向对象程序设计

本书是根据面向对象程序设计课程的基本教学要求,针对已学过C语言程序设计的读者编写的,主要介绍C++面向对象程序设计的基本知识和编程方法,阐述了C++语言实现面向对象基本特性的关键技术。

- 作者希望通过大量例题,以通俗易懂的语言讲解复杂的概念和方法,使读者能深刻理解和领会面向对象程序设计的特点和风格,掌握其方法和要领。
- 作者希望通过精心设计的大量习题和上机实验题,训练学生分析问题、解决问题及编程能力,使学生尽快地迈入面向对象程序设计的大门。

本教材的结构框图



免费提供

PPT等教学相关资料



人民邮电出版社
教学服务与资源网
www.ptpedu.com.cn

教材服务热线: 010-67170985

反馈/投稿/推荐信箱: 315@ptpress.com.cn

人民邮电出版社教学服务与资源网: www.ptpedu.com.cn



封面设计: 董志楠

人民邮电出版社网址: www.ptpress.com.cn

ISBN 978-7-115-22780-5



9 787115 227805 >

ISBN 978-7-115-22780-5

定价: 33.00 元

21世纪高等学校计算机规划教材

21st Century University Planned Textbooks of Computer Science

C++面向对象程序设计

C++ Object-Oriented Programming

陈维兴 陈昕 编著



精品系列

人民邮电出版社

北京

图书在版编目(CIP)数据

C++面向对象程序设计 / 陈维兴, 陈昕编著. — 北京: 人民邮电出版社, 2010.10
21世纪高等学校计算机规划教材
ISBN 978-7-115-22780-5

I. ①C… II. ①陈… ②陈… III. ①C语言—程序设计—高等学校—教材 IV. ①TP312

中国版本图书馆CIP数据核字(2010)第117253号

内 容 提 要

本书介绍了C++面向对象程序设计的基本知识和编程方法,以及C++面向对象的基本特征。针对初学者的特点,本书力求通过大量实例、习题和上机实验题,以通俗易懂的语言讲解复杂的概念和方法,使读者能深刻理解和领会面向对象程序设计的特点和风格,掌握其方法和要领,以期帮助读者尽快地迈入面向对象程序设计的大门。

本书以应用为目的,大力加强实践环节,注重培养应用能力,适合作为高等院校各专业学生学习C++程序设计课程的教材,也可作为C++语言自学者的参考用书。

21世纪高等学校计算机规划教材

C++面向对象程序设计

-
- ◆ 编 著 陈维兴 陈 昕
责任编辑 武恩玉
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市海波印务有限公司印刷
 - ◆ 开本: 787×1092 1/16
印张: 19.75
字数: 521千字
 - 2010年10月第1版
2010年10月河北第1次印刷

ISBN 978-7-115-22780-5

定价: 33.00元

读者服务热线: (010)67170985 印装质量热线: (010)67129223
反盗版热线: (010)67171154

面向对象程序设计是不同于传统程序设计的一种新的程序设计范型。它对降低软件的复杂性,改善其重用性和维护性、提高软件的生产效率,有着十分重要的意义。因此面向对象的程序设计被普遍认为是程序设计方法学的一场实质性的革命。

C++是在 C 语言基础上扩充了面向对象机制而形成的一种面向对象程序设计语言,它除了继承 C 语言的全部优点和功能外,还支持面向对象程序设计。C++现在已成为介绍面向对象程序设计的首选语言。学习 C++不仅可以深刻理解和领会面向对象程序设计的特点和风格,掌握其方法和要领,而且可以使读者掌握一种十分流行和实用的程序设计语言。

目前市场上的 C++教材很多,但大多数教材都是为没有 C 语言基础的学生编写的。据作者了解,当前无论在大学里或社会上,有相当一批人已经学过 C 语言。很多高校的培养计划,仍是先开设 C,随后再开设 C++。本书就是为那些已经学过 C 语言,具有一定程序设计基础的大学本科生编写的。因此,本书是符合高校教学需要的。在取材方面,舍去了 C 语言已经学过的内容,只讲 C++面向对象程序设计部分的内容。这样既节省了教学时间,也减轻了学生的学习负担。根据多年师生反馈的信息,本书的取材是合适的,深度也是适宜的。

本书体现了“以学生为中心”的理念,内容叙述力求通俗易懂,由浅入深,符合认知规律,力求做到多讲实例,循序渐进地引出概念,将复杂的概念用简洁浅显的语言来讲述。力求教学内容富有启发性,便于学生学习。本书以应用为目的,大力加强实践环节,注重培养应用能力。在本书中配有大量的例题、上机实验题和习题,以利于学生举一反三,从中学习方法和技巧,注意培养学生的实践能力和创新能力。

本书共分 11 章。第 1 章介绍面向对象程序设计的基本概念;第 2 章介绍 C++对 C 语言在非面向对象方面的扩充;第 3 章至第 10 章详述了 C++支持面向对象程序设计的基本方法,包括类、对象、派生类、继承、多态性、模板、流类库、异常处理和命名空间等;第 11 章介绍了面向对象程序设计的一般方法和技巧,并安排了一个应用实例,供读者借鉴。在附录中介绍了 Visual C++ 6.0 的开发环境,对如何建立和运行 C++单文件程序和多文件程序进行了较详细的介绍。每一章后面给出了上机实验题和习题,供读者练习。

本书的上机实验部分由陈昕编写,其他章节由陈维兴编写。全书由陈维兴组织编写并统稿。本书中所有程序都经作者在 Visual C++ 6.0 上调试通过。

在本书的编写和出版过程中得到了周涛、林小茶、李春强、孙若莹的帮助和支持,在此表示诚挚的感谢。

最后,借用本书出版的机会,向关心本书的各位专家、老师和广大读者表示衷心的感谢,欢迎您对本书的内容和编写方法提出批评和建议。

编 者

2010 年 6 月

目 录

第 1 章 面向对象方法概述.....1

1.1 什么是面向过程程序设计方法.....1

1.1.1 面向过程程序设计方法概述.....1

1.1.2 面向过程程序设计方法的局限性.....3

1.2 什么是面向对象程序设计方法.....4

1.2.1 面向对象程序设计方法的基本 概念.....4

1.2.2 面向对象程序设计方法的基本 特征.....7

1.2.3 面向对象程序设计方法的主要 优点.....10

1.3 面向对象程序设计语言.....11

1.3.1 面向对象程序设计语言的发展 概况.....11

1.3.2 几种典型的面向对象程序设计 语言.....12

习题.....12

第 2 章 C++的初步知识.....14

2.1 C++的发展和特点.....14

2.1.1 C++的发展.....14

2.1.2 C++的特点.....15

2.2 C++源程序的构成.....15

2.2.1 一个简单的 C++程序.....15

2.2.2 C++程序的结构特性.....18

2.3 C++程序的编辑、编译、连接和 运行.....18

2.4 C++对 C 的扩充.....19

2.4.1 注释.....19

2.4.2 C++的输入输出.....20

2.4.3 灵活的局部变量说明.....23

2.4.4 const 修饰符.....23

2.4.5 函数原型.....24

2.4.6 内联函数.....27

2.4.7 带有默认参数的函数.....28

2.4.8 函数的重载.....29

2.4.9 作用域运算符“::”.....31

2.4.10 强制类型转换.....32

2.4.11 运算符 new 和 delete.....32

2.4.12 引用.....35

实验.....40

习题.....41

第 3 章 类和对象.....46

3.1 类的构成.....46

3.1.1 从结构体到类.....46

3.1.2 类的构成.....47

3.2 成员函数的定义.....49

3.2.1 普通成员函数的定义.....49

3.2.2 内联成员函数的定义.....51

3.3 对象的定义和使用.....52

3.3.1 类与对象的关系.....52

3.3.2 对象的定义.....52

3.3.3 对象中成员的访问.....53

3.3.4 类的作用域和类成员的访问 属性.....55

3.4 构造函数与析构函数.....56

3.4.1 对象的初始化和构造函数.....56

3.4.2 用成员初始化表对数据成员初 始化.....60

3.4.3 析构函数.....61

3.4.4 默认的构造函数和默认的析构 函数.....64

3.4.5 带默认参数的构造函数.....66

3.4.6 构造函数的重载.....67

3.5 对象的赋值与复制.....68

3.5.1 对象赋值语句.....68

3.5.2 拷贝构造函数.....70

3.6 自引用指针 this.....75

3.7 C++的 string 类.....77

3.8 应用举例.....79

实验	80	5.4 多重继承与虚基类	142
习题	82	5.4.1 声明多重继承派生类的方法	143
第4章 类和对象的进一步讨论	87	5.4.2 多重继承派生类的构造函数与析构函数	145
4.1 对象数组与对象指针	87	5.4.3 虚基类	148
4.1.1 对象数组	87	5.5 应用举例	153
4.1.2 对象指针	90	实验	155
4.2 向函数传递对象	92	习题	158
4.2.1 使用对象作为函数参数	92	第6章 多态性与虚函数	163
4.2.2 使用对象指针作为函数参数	93	6.1 多态性概述	163
4.2.3 使用对象引用作为函数参数	93	6.2 基类与派生类对象之间的赋值兼容关系	163
4.3 静态成员	94	6.3 虚函数	166
4.3.1 静态数据成员	95	6.3.1 虚函数的引入	166
4.3.2 静态成员函数	99	6.3.2 虚函数的作用和定义	168
4.4 友元	103	6.3.3 虚析构函数	173
4.4.1 友元函数	103	6.4 纯虚函数和抽象类	175
4.4.2 友元类	106	6.4.1 纯虚函数	175
4.5 类的组合	108	6.4.2 抽象类	176
4.6 共享数据的保护	111	6.5 应用举例	177
4.6.1 常对象	111	实验	179
4.6.2 常对象成员	112	习题	180
4.7 C++的多文件程序	114	第7章 运算符重载	183
4.8 应用举例	116	7.1 运算符重载概述	183
实验	119	7.2 运算符重载函数作为类的友元函数和成员函数	186
习题	121	7.2.1 运算符重载函数作为类的友元函数	186
第5章 继承与派生	126	7.2.2 运算符重载函数作为类的成员函数	190
5.1 继承与派生的基本概念	126	7.2.3 运算符重载应该注意的几个问题	193
5.1.1 为什么要使用继承	126	7.3 前置运算符和后置运算符的重载	196
5.1.2 派生类的声明	128	7.4 重载插入运算符和提取运算符	199
5.1.3 基类成员在派生类中的访问属性	129	7.4.1 重载插入运算符 "<<"	199
5.1.4 派生类对基类成员的访问规则	130	7.4.2 重载提取运算符 ">>"	201
5.2 派生类的构造函数和析构函数	136	7.5 不同类型数据间的转换	203
5.2.1 派生类构造函数和析构函数的调用顺序	136	7.5.1 系统预定义类型间的转换	203
5.2.2 派生类构造函数和析构函数的构造规则	137		
5.3 在派生类中显式访问基类成员	141		

7.5.2 类类型与系统预定义类型间的 转换	204	9.4 应用举例	257
7.6 应用举例	208	实验	259
实验	211	习题	261
习题	212	第 10 章 异常处理和命名空间	264
第 8 章 模板	216	10.1 异常处理	264
8.1 模板的概念	216	10.1.1 异常处理概述	264
8.2 函数模板	217	10.1.2 异常处理的方法	265
8.2.1 函数模板的声明	217	10.2 命名空间和头文件命名规则	269
8.2.2 函数模板的使用	217	10.2.1 命名空间	269
8.3 类模板	221	10.2.2 头文件命名规则	271
8.4 应用举例	227	10.3 应用举例	272
实验	229	实验	273
习题	230	习题	274
第 9 章 C++的输入和输出	233	第 11 章 综合设计与实现	276
9.1 C++流的概述	233	11.1 需求分析	276
9.1.1 C++的输入/输出流	233	11.2 系统分析	276
9.1.2 预定义的流对象	234	11.2.1 基本信息类的属性和操作	276
9.1.3 输入输出流的成员函数	235	11.2.2 各种学生类的属性和操作	277
9.2 预定义类型输入输出的格式控制	237	11.2.3 系统管理类的操作	277
9.2.1 用流成员函数进行输入输出格 式控制	237	11.3 系统设计	278
9.2.2 使用预定义的操纵符进行输入 输出格式控制	241	11.3.1 基类和派生类的设计	278
9.2.3 使用用户自定义的操纵符进行 输入输出格式控制	244	11.3.2 系统管理类的设计	280
9.3 文件的输入输出	245	11.4 系统实现	282
9.3.1 文件的概述	245	实验	291
9.3.2 文件的打开与关闭	246	习题	291
9.3.3 文本文件的读写	249	附录 C++上机操作介绍	292
9.3.4 二进制文件的读写	252	附录 A Visual C++ 6.0 的开发环境	292
		附录 B 建立和运行单文件程序	295
		附录 C 建立和运行多文件程序	303

第 1 章

面向对象方法概述

面向对象程序设计 (Object-Oriented Programming, OOP) 的思想已经被越来越多的软件设计人员所接受。之所以这样, 不仅因为它是一种最先进的、新颖的计算机程序设计思想, 更主要的是这种新的思想更接近人的思维活动, 人们利用这种思想进行程序设计时, 可以很大程度地提高编程能力, 减少软件维护的开销。面向对象程序设计方法是通过增加软件的可扩充性和可重用性来提高程序员的编程能力。这种思想与我们以前使用的方法有很大的不同, 并且在理解上有一些难点, 希望本章的内容能对读者有所帮助。

1.1 什么是面向过程程序设计方法

面向对象程序设计方法和面向过程程序设计方法比较起来, 有许多优越性。那么, 什么是面向对象程序设计方法呢? 在介绍它之前, 首先讨论面向过程程序设计方法。

1.1.1 面向过程程序设计方法概述

面向过程程序设计方法是流行很广泛的程序设计方法。在面向过程程序设计中, 程序设计者不仅要考虑程序要“做什么”, 还要解决“怎么做”的问题。首先要明确程序的功能, 程序设计的重点是如何设计算法和实现算法。在面向过程程序设计中, 一种普遍采用的优化方法是使用结构化程序设计方法, 即所有的程序都可以由顺序、分支和循环 3 种基本结构组成。这样在问题求解过程中, 我们可以先进行整体规划, 将一个复杂的任务按功能分解成一个个易于控制和处理的子任务。然后对每个子任务按功能再进行细化, 依此进行, 直到不需要细分为止。具体实现程序时, 每个子任务对应一个子模块, 模块间尽量相对独立, 通过模块间的调用关系或全局变量而有机地联系起来。

在 C 语言中, 可以将每一个子模块对应设计成一个函数, 各个函数及函数间的调用关系组成了程序。下面, 用面向过程程序设计方法编写一个简单的计算面积的 C 语言程序。

例 1.1 利用面向过程程序设计方法计算圆和三角形的面积。

设圆的半径为 r , 圆周率取 3.14, 则圆面积 cs 的计算公式为 $cs=3.14*r*r$, 设三角形的高为 h , 底为 w , 则三角形面积 ts 的计算公式为 $ts=0.5*h*w$ 。

根据结构化程序设计方法, 该任务有一个主模块 (对应主函数 `main`), 两个子模块, 即计算圆面积的子模块 (对应函数 `circle`) 和计算三角形面积的子模块 (对应函数 `triangle`)。其程序的架构可以用图 1-1 来表示。



图 1-1 例 1.1 的程序框架示意图

下面，用面向过程程序设计方法编写一个简单的计算面积的 C 语言程序。

```

#include<stdio.h>
double circle(double r)                                //定义函数 circle
{ return 3.14*r*r;
}
double triangle(double h, double w)                    //定义函数 triangle
{ return 0.5*h*w;
}
int main()                                              //定义主函数 main
{ double r, h, w;
  double cs, ts;
  printf("Input r, h, w: ");
  scanf("%lf%lf%lf", &r, &h, &w);                    //输入圆的半径和三角形的高和底的值
  cs= circle(r);                                       //调用函数 circle
  ts= triangle(h, w);                                  //调用函数 triangle
  printf("The area of circle is:%f\n", cs);           //输出圆的面积
  printf("The area of triangle is: %f\n", ts);        //输出三角形的面积
  return 0;
}
  
```

程序的一次运行结果如下：

```

Input r, h, w: 10 20 10
The area of circle is:314.000000
The area of triangle is: 100.000000
  
```

这个简单的 C 语言程序设计表示了面向过程程序设计的主要特征：程序由过程定义和过程调用组成（所谓过程，简单地说，就是程序执行某项操作的一段代码，函数是最常用的过程。），从这个意义出发，基于面向过程的程序可以用以下的公式来表述：

程序 = 过程 + 调用

推而广之，如果把函数看成是一种过程，那么面向过程程序设计方法所设计的程序的架构可以用图 1-2 来概括。

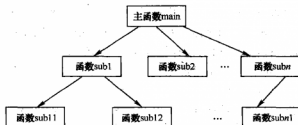


图 1-2 面向过程方法的程序设计框架示意图

结合图 1-2 所示,可以看出:

(1) 面向过程程序设计方法一般采用自上而下、逐步求精的结构化程序设计方法。当编写一个较大的程序时,可以把它按照功能逐级划分成许多相对独立的小模块。每个小模块的功能由一个函数实现,再通过适当的方法将这些函数组织在一起协同工作,就能够完成整个程序所规定的任务。

(2) 使用面向过程程序设计方法编写出的 C 语言程序包括一个主函数 `main` 和若干用户定义函数。主函数由操作系统调用,它是整个程序的入口。主函数的主要任务就是完成对其他函数的有机调用。其他函数之间也可以相互调用,并且同一个函数可以被一个或多个函数调用任意多次。

1.1.2 面向过程程序设计方法的局限性

众所周知,计算机的发明和使用,对人类社会的进步与发展发挥了巨大的作用。随着计算机大规模地推广、普及与应用,人们对软件的功能要求越来越多,对软件的性能要求越来越高。传统的程序设计已不能满足这些日益增长的需要。面向过程程序设计中普遍采用的优化方法是使用结构化程序设计方法,其局限性体现在以下几个方面。

1. 面向过程程序设计方法开发软件的生产效率低下

按照结构化程序设计的要求,当需要解决一个复杂的任务时,首先应将它按功能划分为若干个小任务,每个小任务又可以按功能划分为若干个更小的任务,依次类推……直到最低一层的任务容易用程序实现为止,然后将所有的小任务全部解决并把它们组合起来。

这种传统程序设计的生产方式仍是采用较原始的方式进行,程序设计基本上还是从语句一级开始。软件的生产中缺乏大粒度、可重用的构件,软件的重用问题没有得到很好地解决,而导致软件生产的工程化和自动化屡屡受阻。

面向过程程序设计的特点是数据与其操作分离,而且对同一数据的操作往往分散在程序的不同地方。这样,如果一个或多个数据的结构发生了变化,那么这种变化将波及程序的很多部分甚至遍及整个程序,致使许多函数必须重写,严重时会导致整个软件结构的崩溃。数据和操作相分离的结构,使得维护数据和处理数据的操作过程要花费大量的精力和时间,严重地影响了软件的生产效率。

2. 面向过程程序设计方法难以应付日益庞大的信息量和多样的信息类型

随着计算机科学与技术的飞速发展和计算机应用的普及,当代计算机的应用领域已从数值计算扩展到了人类社会的各个方面,所处理的数据已从简单数字和字符发展为具有多种格式的多媒体数据,如文本、图形、图像、影像、声音等,描述的问题已从单纯的计算问题发展到仿真复杂的自然现象和社会现象。于是,计算机处理的信息量与信息类型迅速增加,程序的规模日益庞大,复杂度不断增加。这些都要求程序设计语言有更大的信息处理能力。然而,面对这些庞大的信息量和多样的信息格式,面向过程程序设计方法是无法应付的。

3. 面向过程程序设计方法难以适应各种新环境

当前,并行处理、分布式、网络和多机系统等,已经或将是程序运行的主流方式和主流环境。这些环境的一个共同特点是都具有一些有独立处理能力的节点,节点之间有通信机制,即以消息传递进行联络。显然传统的程序设计技术很难适应这些新环境。

综上所述,面向过程程序设计方法不能够满足计算机技术迅猛发展的需要,软件开发迫切需要一种新的程序设计方法的支持。那么,面向对象程序设计是否能够担当此任呢?下面

我们再分析一下面向对象程序设计的方法。

1.2 什么是面向对象程序设计方法

1.2.1 面向对象程序设计方法的基本概念

为了理解面向对象程序设计方法，我们从最基本的概念入手。本节介绍的内容是面向对象程序设计的理论基础，这些内容不依赖于具体的程序设计语言，也就是说，无论使用哪种面向对象语言进行面向对象程序设计，本节内容都有理论意义。

1. 对象

面向对象程序设计方法是一种自下而上的程序设计方法，它不像面向过程程序设计方法那样，需要在开始就要用主函数 main 概括出整个程序，它往往是从问题的一部分着手，一点一点地构建出整个程序。面向对象程序的基本元素是对象。

对象就是可以控制和操作的实体。在现实世界中，任何事物都是对象。它可以是一个有形的、具体存在的事物，例如一张桌子、一个学生、一辆汽车甚至一个地球。它也可以是一个无形的、抽象的事件，例如一次演出、一场球赛、一次出差等。对象既可以很简单，也可以很复杂，复杂的对象可以由若干简单的对象构成，整个世界都可以认为是一个非常复杂的对象。

现实世界中的对象既具有静态的属性（或称状态），又具有动态的行为（或称功能）。例如，一个人就是一个对象，每个人都有姓名、性别、年龄、身高、体重等属性，都有吃饭、走路、睡觉、学习等行为。所以在现实世界中，任何一个对象都具有这两个要素，即属性和行为，它能根据外界给出的信息进行相应的操作。一个对象往往是由一组属性和一组行为构成的。

现实世界中的对象，具有以下特性：

- （1）有一个名字：每一个对象必须有一个名字，称为对象名，以区别于其他对象；
- （2）有一组属性：用属性来描述它的某些特征，一般可以用数据来表示，所有的属性都有值；
- （3）有一组行为：对象的行为或功能也称为方法，一般用一组操作来描述；
- （4）有一组接口：除施加于对象内部的操作外，对象还提供了一组公有操作作用做与外界的接口，从而可以与其他对象建立关系。

面向对象的程序设计采用了以上人们所熟悉的这种思路。在面向对象程序设计中，对象是描述其属性的数据以及对这些数据施加的一组操作封装在一起构成的统一体。在 C++ 语言中，每个对象都是由数据（属性）和操作代码（行为，通常用函数来实现）两部分组成的，如图 1.3 所示。在面向对象程序设计中，用数据来体现上面提到的“属性”，如一个圆对象，它的半径就是它的属性。函数是用来对数据进行操作的，以实现某些功能。例如，可以通过半径计算出圆的面积，并且输出圆的半径和面积。计算圆的面积和输出有关数据就是前面提到的行为。

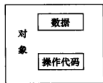


图 1-3 对象结构示意图

2. 类

在实现世界中,“类”是一组具有相同属性和行为的对象的抽象。例如,虽然张三、李四、王五……,每个人的性格、爱好、职业、特长等各有不同,但是他们的基本特征是相似的,都具有相同的生理构造,都能吃饭、说话、走路等,于是把他们统称为“人”类,而具体的每一个人是人类的一个实例,也就是一个对象。

类在现实世界中并不真正存在。例如,在地球上并没有抽象的“学生”,只有一个个具体的学生,如张三、李四、王五……。同样,世界上也没有抽象的“教师”,只有一个个具体的教师。

在面向对象程序设计中,“类”就是具有相同的数据(属性)和相同的操作代码(函数)的一组对象的集合。对象是具体存在的,如一个圆可以作为一个对象,10个不同半径的圆就是10个对象。如果这10个圆对象有相同的数据(属性)和操作代码(函数),就可以将它们抽象为一种类型,称为圆类型。在C++语言中,可以将这种类型定义为类(class)。需要说明的是,这里所说的数据相对于现实世界中属性,具体指圆的半径,各个圆对象的半径值可以不同。在C++语言中把类中的数据称为数据成员,类中的操作是用函数来实现的,这些函数称为成员函数。

类和对象之间的关系是抽象和具体的关系。类是多个对象进行综合抽象的结果,一个对象是类的一个实例。例如“学生”是一个类,它是由许多具体的学生抽象而来的一般概念。同理,桌子、教师、计算机等都是类。

在面向对象程序设计中,总是先声明类,再由类生成其对象。类是建立对象的“模板”,按照这个模板所建立的一个个具体的实例,通常称为对象。打个比方,手工制作月饼时,先雕刻一个有凹下图案的木模,然后将事先揉好的面塞进木模里,用力挤压后,将木模反扣在桌上,一个漂亮的图案就会出现在月饼上了。这样一个接着一个就可以制造出外形一模一样的月饼,但每个月饼中的面和馅的量可以是不同的。这个木模就好比是“类”,制造出来的糕点好比是“对象”。

3. 消息

现实世界中的对象不是孤立存在的实体,它们之间存在着各种各样的联系,正是它们之间的相互作用、联系和连接,才构成了世间各种不同的系统。

在面向对象程序设计中,对象之间也需要联系,我们称为对象的交互。面向对象程序设计技术必须提供一种机制允许一个对象与另一个对象的交互,这种机制称为消息传递。一个对象向另一个对象发出的请求称为“消息”。当对象接收到发向它的消息时,就调用有关的方法,执行相应的操作。例如,有一个教师对象张三和一个学生对象李四,对象李四可以发出消息,请求对象张三演示一个实验,当对象张三接收到这个消息后,确定应完成的操作并执行之。

一般情况下,我们称发送消息的对象为发送者或请求者,接收消息的对象为接收者或目标对象。发送者发送消息,接收者通过调用相应的方法对消息作出响应。这个过程不断重复,系统不停地运转,最终得到相应的结果。

一个对象可以同时向多个对象发送消息,也可以接收多个对象发来的消息。相同形式的消息可以发送给不同的对象,同一个对象也可以接收不同形式的消息。消息的发送者可以不考虑具体接收者,只是反映了发送者的请求。由于消息的识别、解释取决于接收者,对象可以响应消息也可以不响应消息。

4. 方法

在面向对象程序设计中的消息传递实际是对现实世界中的信息传递的直接模拟。调用对象中的函数就是向该对象传送一个消息，要求该对象实现某一行为（功能）。对象所能实现的行为（功能），在程序设计方法中称为方法，它们是通过调用相应的函数来实现的。在 C++ 语言中，方法是通过成员函数来实现的。

方法包括界面和方法体两部分。方法的界面给出了方法名和调用协议（相对于 C++ 语言中成员函数的函数名和参数表）；方法体则是实现某种操作的一系列计算步骤，也就是一段程序（相对于 C++ 语言中成员函数的函数体）。消息和方法的关系是：对象根据接收到的消息，调用相应的方法；反过来，有了方法，对象才能响应相应的消息。

面向对象程序设计是一种新的程序设计范型。这种范型的主要特征如下：

程序 = 对象 + 消息

对于面向对象的程序设计，程序员注重的是类的设计和编写，即问题域中涉及几个类，各个类之间的关系如何，每个类包含哪些数据和函数（操作代码），再由类生成其对象。程序中的一切操作都是通过向对象发送消息来实现的，对象接收到消息后，启动有关方法（通过成员函数）完成相应的操作。

下面我们将例 1.1 用 C++ 面向对象程序设计语言进行重新编写。在此，读者不需要完全理解，只要有一个初浅的了解即可。

例 1.2 利用面向对象思想求解计算圆和三角形的面积。

首先利用面向对象的思想来分析这个问题，本题可声明两个类：圆类 Circle 和三角形类 Triangle。

圆类 Circle 中含有数据成员（属性）r 和 cs，r 和 cs 分别表示圆的半径和圆的面积；含有成员函数（操作代码）radius_input 和 c_area_out，成员函数 radius_input 用于输入圆半径 r 的值，成员函数 c_area_out 用于计算与输出圆面积 cs 的值。由圆类 Circle 定义的各种大小不同的圆都是圆类的对象。

三角形类 Triangle 中含有数据成员（属性）h、w 和 ts，h 表示三角形的高，w 表示三角形的底边，ts 表示三角形的面积；含有成员函数（操作代码）h_w_input 和 t_area_out，成员函数 h_w_input 用于输入高 h 与底边 w 的值，成员函数 t_area_out 用于计算与输出三角形面积 ts 的值。由三角形类 Triangle 定义的各种大小和形状不同的三角形都是三角形类的对象。

```
#include<stdio.h>
class Circle {
    double r;           //声明圆类 Circle
    double cs;          //定义数据成员 r(表示圆的半径)
    public:
    void radius_input()  //定义数据成员 cs(表示圆的面积)
    { printf("Input r : ");
      scanf("%lf", &r);
    }
    void c_area_out()    //定义成员函数 radius_input (用于输入圆半径 r 的值)
    { cs=3.14*r*r;
      printf("The area of circle is:%f\n", cs);
    }
};
//圆类 Circle 到此结束
```

```

class Triangle{           //声明三角形类 Triangle
    double h, w;          //定义数据成员 h 和 w (表示三角形的高和底边的值)
    double ts;            //定义数据成员 ts (表示三角形的面积)
public:
    void h_w_input()       //定义成员函数 h_w_input (用于输入三角形的高 h 与底边 w 的值)
    { printf("Input h, w: ");
      scanf("%lf%lf", &h, &w);
    }
    void t_area_out()      //定义成员函数 t_area_out (用于计算与输出三角形面积)
    { ts=0.5*h*w;
      printf("The area of triangle is: %f\n", ts);
    }
};                          //三角形类 Triangle 到此结束

int main()                //定义主函数 main
{ Circle c1;              //定义圆类 Circle 的对象 c1
  c1.radius_input();       //向对象 c1 传送一个消息, 即调用函数 radius_input, 输入圆半径 r 的值
  c1.c_area_out();         //向对象 c1 传送一个消息, 即调用函数 c_area_out
                          //计算与输出圆面积

  Triangle t1;            //定义三角形类 Triangle 的对象 t1
  t1.h_w_input();          //向对象 t1 传送一个消息, 即调用函数 h_w_input, 输入三角形的高 h 与底边 w 的值
  t1.t_area_out();         //向对象 t1 传送一个消息, 即调用函数 t_area_out, 计算与输出三角形面积
  return 0;
}                          //主函数 main 到此结束

```

程序的一次运行结果如下:

```

Input r : 10
The area of circle is:314.000000
Input h, w: 10 20
The area of triangle is: 100.000000

```

为了使程序简单明了, 以上程序只定义了一个圆类对象 c1 和一个三角形对象 t1。实际使用时, 可根据需要定义多个对象。读者可以通过例 1.1 和例 1.2 的对比分析, 初步了解面向过程设计方法和面向对象程序设计方法的区别。

需要说明的是, 基于面向过程程序设计方法的语言称为面向过程性语言, 如 C 语言等就是典型的面向过程性语言。但是, 某一种程序设计语言不一定与一种程序设计方法相对应。实际上存在具有两种或多种程序设计方法的程序设计语言, 即混合型语言。例如, C++ 语言就是具有面向过程程序设计方法和面向对象程序设计方法的混合型程序设计语言。

1.2.2 面向对象程序设计方法的基本特征

面向对象程序设计方法模拟人类习惯的解题方法, 代表了计算机程序设计的新颖的思维方法。这种方法的提出是对软件开发方法的一场革命, 是目前解决软件开发面临困难的最有希望、最有前途的方法之一。本节介绍面向对象程序设计的 4 个基本特征。

1. 抽象

抽象是人类认识问题的最基本的手段之一。抽象是将有关事物的共性归纳、集中的过程。

在现实生活中，人们能看到的都是一些具体的事物，例如男人、女人、大人、小孩等，这些都是具体的人，把这些具体的人归纳为一类，称为“人”，这就是一种抽象。再假如，把所有具有大学学籍的人归为一类，称为“大学生”，这也是一个抽象。抽象是对复杂世界的简单表示，抽象并不打算了解全部问题，而只强调感兴趣的信息，忽略了与主题无关的信息。例如，在设计一个学生成绩管理系统的过程中，只关心学生的姓名、学号、成绩等，而对学生的身高、体重等信息就可以忽略。而在学生健康信息管理系统中，身高、体重等信息必须抽象出来，而成绩则可以忽略。

在面向对象程序设计中，抽象是通过特定的实例（对象）抽取共同特性后形成概念的过程。C和C++语言中的基本数据类型就是对一批具体的数的抽象。例如，“整型数据”是对所有整数的抽象。

2. 封装

在现实世界中，封装就是把某个事物包围起来，使外界不知道该事物的具体内容。在面向对象程序设计中，封装是指把数据和实现操作的代码集中起来放在对象内部，并尽可能隐蔽对象的内部细节。对象好像是一个不透明的黑盒子，表示对象属性的数据和实现各个操作的代码都被封装在黑盒子里，从外面是看不见的，更不能从外面直接访问或修改这些数据及代码。使用一个对象的时候，只需知道它向外界提供的接口而无须知道它的数据结构细节和实现操作的算法。这很像电视机、录音机、洗衣机等，从其外形来看，都是各种电子或机械部件被封装在盒子内部。使用这些电器的人并不需要知道电器内部有哪些部件，它们是如何组装的，它们的工作原理又如何。使用者只需要会使用电器提供的几个外部按钮（对应于对象的外部接口），就可以实现自己所需要的功能。将电器部件封装在盒子内部，既可以避免各种人为的损坏，也便于维护和管理。

C++程序中，对象的函数名就是对象的对外接口，外界可以通过函数名来调用这些函数来实现某些行为（功能）。这些将在以后详细介绍。

封装的好处是可以将对象的使用者与设计者分开，大大降低了人们操作对象的复杂程度。使用者不必知道对象行为实现的细节，只需要使用设计者提供的接口的功能，即可自如地操作对象。封装的结果实际上隐蔽了复杂性，并提供了代码重用性，从而减轻了开发一个软件系统的难度。

封装是面向对象程序设计方法的一个重要特性。封装具有两方面的含义：一是将有关的数据和操作代码封装在一个对象中，各个对象相对独立、互不干扰；二是将对象中某些数据与操作代码对外隐蔽，即隐蔽其内部细节，只留下少量接口，以便与外界联系，接收外界的消息。这种对外界隐蔽的做法称为信息隐蔽。信息隐蔽有利于数据安全，防止无关人员访问和修改数据。

3. 继承

继承在现实生活中是一个很容易理解的概念。例如，我们每一个人都从我们的父母身上继承了一些特性，如种族、血型、眼睛的颜色等，我们身上的特性来自我们的父母，也可以说，父母是我们所具有的属性和行为的基础。

利用继承可以简化人们对事物的认识和叙述，简化工作程序。例如“柯利狗”继承了“狗”的特性，现在要叙述“柯利狗”的特征，显然不需要从头介绍什么是狗，而只要说明“柯利狗是尖鼻子、红白相间的颜色、适合放牧的狗”即可。利用继承可以简化程序设计的步骤。例如，在软件开发中已经建立了一个类A，又需要建立另一个与A基本相同，但增加了一些

属性和行为的类 B, 这时没有必要从头设计一个新类, 只需在类 A 的基础上增加一些新的内容即可。这就是面向对象程序设计中的继承机制。

以面向对象程序设计的观点, 继承所表达的是类之间相关的关系。这种关系使得某一类可以继承另外一个类的特征和能力。

若类之间具有继承关系, 则它们之间具有下列几个特性:

- (1) 类间具有共享特征 (包括数据和操作代码的共享);
- (2) 类间具有差别或新增部分 (包括非共享的数据和操作代码);
- (3) 类间具有层次结构。

假设有两个类 A 和 B, 若类 B 继承类 A, 则类 B 包含了类 A 的特征 (包括数据和操作), 同时也可以加入自己所特有的新特性。这时, 我们称被继承类 A 为基类或父类; 而称继承类 B 为 A 的派生类或子类。同时, 我们还可以说, 类 B 是从类 A 中派生出来的。

如果类 B 是类 A 的派生类, 那么, 在构造类 B 的时候, 我们不必描述 B 的所有特征, 我们只需让它继承类 A 的特征, 然后描述与基类 A 不同的那些特性。也就是说, 类 B 的特征由继承来的和新添加的两部分特征构成。

具体地说, 继承机制允许派生类继承基类的数据和操作 (即数据成员和成员函数), 也就是说, 允许派生类使用基类的数据和操作。同时, 派生类还可以增加新的操作和数据。

采用继承的方法可以很方便地利用一个已有的类建立一个新的类, 这就可以重用已有软件中的一部分甚至大部分, 在派生类中只需描述其基类中没有的数据和操作。这样, 就避免了公用代码的重复开发, 增加了程序的可重用性, 减少了代码和数据冗余, 大大节省了编程的工作量, 这就是常说的“软件重用”思想。同时, 在描述派生类时, 程序员还可以覆盖基类的一些操作, 或修改和重定义基类中的操作。具体的实现方法将在以后详细介绍。

从继承源来分, 继承分为单继承和多继承。

单继承是指每个派生类只直接继承了一个基类的特征。例如, 图 1-4 表示了一种单继承关系, 它表示 Windows 操作系统窗口之间的继承关系。

单继承并不能解决继承中的所有问题, 例如, 小孩喜欢的玩具车既继承了车的一些特性, 也继承了玩具的一些特征, 这就是多继承的一个例子, 如图 1-5 所示。



图 1-4 单继承示意图



图 1-5 多继承示意图

多继承是指多个基类派生出一个派生类的继承关系。多继承的派生类直接继承了不少一个基类的特征。

4. 多态

多态性也是面向对象系统的重要特性。在讨论面向对象程序设计的多态性之前, 我们还是来看看现实世界的多态性。现实世界的多态性在自然语言中经常出现。假设一辆汽车停在了属于别人的车位, 司机可能会听到这样的要求: “请把你的车挪开”。司机在听到请求后, 所做的工作应该是把车开走。在家里, 一把凳子挡住了孩子的去路, 他可能会请求妈妈“请

把凳子挪开”，妈妈过去搬起凳子，放在一边。在这两件事情中，司机和妈妈的工作都是“挪开”一样东西，但是他们在听到请求以后的行为是截然不同的，这就是现实世界中的多态性。对于“挪开”这个请求，还可以有更多的行为与之对应。“挪开”从字面上看是相同的，但由于作用的对象不同，操作的方法也就不同。

面向对象程序设计借鉴了现实世界的多态性。面向对象系统的多态性是指不同的对象收到相同的消息时执行不同的操作。例如，有一个窗口（Window）类对象，还有一个棋子（Piece）类对象，当我们对它们发出“移动”的消息时，“移动”操作在 Window 类对象和 Piece 类对象上可以有不同的行为。

C++语言支持两种多态性，即编译时的多态性和运行时的多态性。编译时的多态性是通过函数重载（包括运算符重载）来实现的，运行时的多态性是通过虚函数来实现的。本书将分别在第2章、第6章和第7章对函数重载、虚函数和运算符重载进行详细介绍。

多态性增强了软件的灵活性和重用性，为软件的开发与维护提供了极大的便利。尤其是采用了虚函数和动态联编机制后，允许用户以更为明确、易懂的方式去建立通用的软件。

1.2.3 面向对象程序设计方法的主要优点

面向对象程序设计方法是软件开发史上的一个重要里程碑。这种方法从根本上改变了人们以往设计软件的思维方式，从而使程序设计者摆脱了具体的数据格式和过程的束缚，将精力集中于要处理对象的设计和 research 上，极大地减少了软件开发的复杂性，提高了软件开发的效率。面向对象程序设计主要具有以下优点。

1. 可提高程序的重用性

面向对象程序设计方法能比较好地解决软件重用的问题。对象所固有的封装性和信息隐藏等机理，使得对象内部的实现与外界隔离，具有较强的独立性，它可以作为一个大粒度的程序构件，供同类程序直接使用。

有两种方法可以重复使用一个对象类：一种方法是建立在各种环境下都能使用的类库，供相关程序直接使用；另一种方法是使它派生出一个满足当前需要的新类。继承性机制使得子类不仅可以重用其父类的数据和程序代码，而且可以在父类代码的基础上方便地修改和扩充，这种修改并不影响对原有类的使用。

2. 可控制程序的复杂性

面向对象程序设计方法采用了封装和信息隐藏技术，把数据及对数据的操作放在一个个类中，作为相互依存、不可分割的整体来处理。这样，在程序中任何要访问这些数据的地方都只需简单地通过传递信息和调用方法来进行，这就有效地控制了程序的复杂性。

3. 可改善程序的可维护性

在面向对象程序设计方法中，对对象的操作只能通过消息传递来实现，所以只要消息模式即对应的方法界面不变，方法体的任何修改不会导致发送消息的程序修改，这显然对程序的维护带来了方便。另外，类的封装和信息隐藏机制使得外界对其中的数据 and 程序代码的非法操作成为不可能，这也就大大地减少了程序的错误率。

4. 能够更好地支持大型程序设计

类是一种抽象的数据类型，所以类作为一个程序模块，要比通常的子程序的独立性强得多，面向对象技术在数据抽象上又引入了动态连接和继承性等机制，进一步发展了基于数据抽象的模块化设计，使其更好地支持大型程序设计。

5. 增强了计算机处理信息的范围

面向对象程序设计方法把描述事物静态属性的数据结构和表示事物动态行为的操作放在一起构成一个整体,完整地、自然地表示客观世界中的实体。用类来直接描述现实世界中的类型,可使计算机系统的描述和处理对象从数据扩展到现实世界和思维世界的各种事物,这实际上大大扩展了计算机系统处理的信息量和信息类型。

由于面向对象程序设计方法的上述优点,我们看到:面向对象程序设计方法是目前解决软件开发面临难题的最有希望、最有前途的方法之一。

1.3 面向对象程序设计的语言

1.3.1 面向对象程序设计语言的发展概况

为了适应高科技发展的需要,为了消除传统程序设计的局限性,自20世纪70年代以来研制出了各种不同的面向对象程序设计语言。现在公认的第一个真正面向对象程序设计语言是Smalltalk。它是由美国的Xerox公司于20世纪70年代初研制的。该语言第一次使用了“面向对象”的概念和程序风格,开创了面向对象程序设计的新范型,被誉为面向对象程序设计语言发展的里程碑。

实际上,面向对象语言的出现并非偶然,它是程序设计语言发展的必然结果。事实上,20世纪60年代研制出来的Simula语言已经引入了几个面向对象程序设计中的概念和特性。Smalltalk中类和继承的概念就是源于Simula语言,它的动态联编(聚束)的概念和交互式开发环境的思想则来自于20世纪50年代诞生的LISP语言,其信息隐藏与封装机制则可以看作是20世纪70年代出现的CLU语言、Modula-2语言及Ada语言数据抽象机制的进一步发展。

Smalltalk的问世,标志着面向对象程序设计语言的正式诞生。20世纪80年代以来,面向对象语言得到飞速发展,形形色色的面向对象语言如雨后春笋般地出现。这时候,面向对象程序设计语言朝着两个方向发展:一个方向是朝着纯面向对象语言发展,如继Smalltalk之后,又出现了Eiffel、SELF等语言;另一个方向是朝着混合型面向对象语言发展,如将过程型与面向对象结合产生了诸如C++、Objective-C、Object Pascal、Object Assembler、Object logo等一大批语言,将函数型(LISP)与面向对象结合产生了诸如LOOPS、Flavors、CLOS等语言,将逻辑型(PROLOG)与面向对象结合产生了诸如SPOOL、Orient 84K等语言。此外,还有一批面向对象的并发程序设计语言也相继出现,如ABCL、POOL、PROCOL等。我们将要学习的C++就是一种面向过程与面向对象相结合的语言。

当前新推出的程序设计语言和软件平台几乎都是面向对象的或基于对象的。例如,我们熟知的Builder C++、Visual C++、Visual Basic、Power Builder、Windows 2000、Windows NT以及现在流行的网上编程语言Java等。这些语言和软件平台把OOP的概念和技术与数据库、多媒体、网络等技术融为一体,成为新一代的软件开发工具与环境。它们的出现标志着OOP已全面进入软件开发的主战场,成为软件开发的主力军。

1.3.2 几种典型的面向对象程序设计语言

1. Smalltalk 语言

Smalltalk 是公认的第一个真正的面向对象程序设计语言,它体现了纯正的面向对象程序设计思想。Smalltalk 中的一切元素都是对象,如数字、符号、串、表达式、程序等都是对象。类也是对象,类是元类的对象。该语言从本身的实现和程序设计环境到所支持的程序设计风格都是面向对象的。

Smalltalk 被认为是最纯正、最具有代表性的面向对象程序设计语言。它在面向对象程序设计乃至面向对象技术中扮演着不可取代的重要角色。

2. Simula 语言

Simula 语言是 20 世纪 60 年代开发出来的,在 Simula 中已经引入了几个面向对象程序设计语言中最重要的概念和特性,如数据抽象的概念、类机构和继承性机制。Simula 67 是具有代表性的一个版本,20 世纪 70 年代发展起来的 CLU、Ada、Modula-2 等语言是在它基础上发展起来的。

3. C++语言

为了填补传统的面向过程程序设计与面向对象程序设计之间的鸿沟,使得人们能从习惯了的面向过程程序设计平滑地过渡到面向对象程序设计,人们对广泛流行的 C 语言进行扩充,C++语言应运而生。我们将在以后的章节进行详细介绍。

4. Java 语言

Java 语言是由 SUN 公司的 J.Gosling、B.Joe 等人在 20 世纪 90 年代初开发出的一种面向对象的程序设计语言。Java 是一个广泛使用的网络编程语言。首先,作为一种程序设计语言,它简单、面向对象、不依赖于机器结构,具有可移植性、鲁棒性和安全性,并且提供了并发的机制,具有很高的性能;其次,它最大限度地利用了网络,Java 的应用程序(Applet)可在网络上传输;另外,Java 还提供了丰富的类库,使程序设计者可以很方便地建立自己的系统。

5. C#语言

C#语言是由 Microsoft 公司于 2000 年 6 月 26 日对外正式发布的。C#语言从 C/C++语言继承发展而来,是一个全新的、面向对象的、现代的编程语言。C#语言可以使广大程序员更容易地建立基于 Microsoft.NET 平台,以 XML(扩展标识语言)为基础的因特网服务应用程序。用 C#语言编写的应用程序可以充分利用 .NET 框架体系带来的各种优点,完成各种各样高级的功能,例如用来编写基于通用网络协议的 Internet 服务软件,也可以编写 Windows 图形用户界面程序,还可以编写各种数据库、网络服务应用程序。

习 题

- 【1.1】面向过程程序设计的主要特征是什么?
- 【1.2】面向过程程序设计方法的局限性至少有哪几个方面?
- 【1.3】什么是面向对象程序设计?
- 【1.4】现实世界中的对象有哪些特征?

- 【1.5】在面向对象程序设计中，什么是对象？什么是类？对象与类的关系是什么？
- 【1.6】什么是消息？
- 【1.7】什么是方法？在 C++ 中它是通过什么来实现的？
- 【1.8】什么是抽象和封装？
- 【1.9】什么是继承性？若类之间具有继承关系，则它们之间具有哪些特性？
- 【1.10】什么是单继承、多继承？请举例说明。
- 【1.11】什么是多态性？请举例说明。
- 【1.12】面向对象程序设计的优点主要有哪些？

第2章

C++的初步知识

C++语言是在 C 语言基础上扩充了面向对象机制而形成的一种面向对象程序设计语言，C++对 C 语言的扩充，主要是引进了面向对象机制。在传统的非面向对象方面，C++对 C 语言也做了不少扩充。本章先介绍这方面的内容，以便为后续章节的学习和编程作好准备。

2.1 C++的发展和特点

2.1.1 C++的发展

C++是从 C 语言发展演变而来。C 语言是 1972 年由 D.M. Ritchie 在美国贝尔实验室设计的一个通用目的程序设计语言，它的前身是 B 语言。C 语言具有许多优点，如语言简洁灵活，运算符和数据结构丰富，具有结构化控制语句，程序执行效率高，同时具有高级语言与汇编语言的优点等。但是 C 语言也存在着一些局限性，如 C 语言的类型检查机制相对较弱，这使得程序中的一些错误不能在编译阶段由编译器检查出来；C 语言本身几乎没有支持代码重用的语言结构；C 语言是一个面向过程的编程语言，不能满足运用面向对象方法开发软件的需要。C 语言不适合开发大型程序，当程序的规模达到一定的程度时，程序员就很难控制程序的复杂性。

C++正是为了解决 C 语言的上述问题而设计的。C++是美国贝尔实验室的 Bjarne Stroustrup 博士及其同事于 20 世纪 80 年代初在 C 语言的基础上开发成功的。C++继承了 C 语言的原有精髓，如高效率、灵活性；扩充增加了对开发大型软件颇为有效的面向对象机制；弥补了 C 语言不支持代码重用、不适宜开发大型软件的不足，成为一种优秀的既支持面向过程程序设计，又支持面向对象程序设计的混合型的程序设计语言。

最初的 C++被称为“带类的 C”，1983 年正式取名为 C++，这是为了强调它是 C 的增强版，采用了 C 语言中的自加运算符“++”。1985 年由 Bjarne Stroustrup 编写的《C++程序设计语言》一书出版，标志着 C++ 1.0 版本的诞生。此后，贝尔实验室又分别推出了 C++ 2.0 版本、C++ 3.0 版本和 C++ 4.0 版本。

C++的标准化工作从 1989 年开始，于 1994 年制定了 ANSI C++标准草案，以后又经过不断完善，于 1998 年 11 月被国际标准化组织（ISO）批准为国际标准（ISO/IEC 14882）。C++就是这样在不断的完善中走过了二十多年的历史。至今，它仍然是一种充满活力的程序设计语言。

2.1.2 C++的特点

C++的主要特点表现在两个方面，一是全面兼容 C 语言，并对 C 语言的功能作了不少扩充，二是增加了面向对象的机制。具体表现如下。

(1) C++是 C 的超集，C++保持与 C 语言的兼容，这就使许多 C 代码不经修改就可以为 C++所用，用 C 编写的众多的库函数和实用软件基本上可以不加修改地用于 C++。

(2) C++是一个更好的 C 语言，它保持了 C 语言的简洁、高效和接近汇编语言等特点，并对 C 语言的功能作了不少扩充。用 C++编写的程序比 C 语言更安全，可读性更好，代码结构更为合理。

(3) 用 C++编写的程序质量高，从开发时间、费用到形成的软件的可重用性、可扩充性、可维护性和可靠性等方面都有了很大的提高，使得大中型的程序开发变得更加容易。

(4) 增加了面向对象的机制，C++几乎支持所有的面向对象程序设计特征，体现了程序设计和软件开发领域出现的新思想和新技术。

C++最有意义的方面是支持面向对象的特征，然而，由于 C++与 C 语言保持兼容，使得 C++不是一个纯正的面向对象的语言，C++既可用于面向过程的结构化程序设计，也可用于面向对象的程序设计。如果读者已经有 C 语言或其他面向过程高级语言的编程经验，那么学习 C++语言时应该着重学习它的面向对象的特征。本书着重介绍 C++面向对象程序设计的基本知识。

2.2 C++源程序的构成

2.2.1 一个简单的 C++程序

下面我们给出一个简单的两数相加的 C++程序，以便读者对 C++程序的格式有一个初步的了解。

例 2.1 计算两个整数之和。

```
//sum.cpp
#include<iostream>           //编译预处理命令
using namespace std;         //使用命令空间 std
int main()                   //主函数首部
{ int x, y, sum;              //定义 3 个整型变量
    cout<<"Please input two integers:"<<"\n"; //提示用户由键盘输入两个整数
    cin>>x;                   //从键盘输入变量 x 的值
    cin>>y;                   //从键盘输入变量 y 的值
    sum=x+y;                  //将 x+y 的值赋给整型变量 sum
    cout<<"x+y="<<sum<<endl; //输出两个整数的和 sum
    return 0;                 //如程序正常结束，向操作系统返回一个数值 0
}
```

本程序用来计算两个整数的和。这个程序非常简单，是一个完整的 C++程序。下面对它进行逐行解释，请读者分析一下，本程序和以前见过的程序有什么不同？

第1行: //sum.cpp

这是一个 C++风格的注释行, 它由 “//” 开始, 到行尾结束, 这条注释行注明了本程序的文件名为 sum.cpp。在程序中可以看到, 以 “//” 开头的注释可以不单独占一行, 它可以出现在一行中的语句之后, 编译器将 “//” 以后到本行末尾的所有字符都作为注释。

第2行: #include <iostream>

这是一条 C++预处理命令, 用来指示编译器在对程序进行预处理时, 将文件 iostream 的代码嵌入到程序中该指令所在的地方。iostream 是一个 C++标准头文件, 其中定义了一些输入输出流对象。

第3行: using namespace std;

这是一条使用命名空间 std 的指令。我们将在后面进行介绍。

第4行: int main()

本行是主函数的声明。主函数是所有 C++程序开始执行的入口。按照 C++的规定, 每个程序必须有且仅有一个主函数, 主函数的名称必须为 main。

标准 C++要求在主函数 main 前面写上返回值类型为 int。若一个函数没有指出返回值类型, C++默认该函数的返回值类型是 int 型。

第5行: int x, y, sum;

这条语句表示, 定义 3 个整型变量 x、y 和 sum。

第6行: cout<<"Please input two integers:"<<"\n";

本行 cout 和 “<<” 的作用是将字符串 “Please input two integers:” 在屏幕上显示出来, “\n” 是换行符, 即输出上述信息后回车换行。

第7行: cin>>x;

和

第8行: cin>>y;

这两行中 cin 和 “>>” 的作用是, 把从键盘输入的两个整数值分别赋给变量 x 和 y。

第9行: sum=x+y;

本行的作用是, 通过赋值语句来计算 x 与 y 的和, 并将 x+y 的值赋给整型变量 sum。

第10行: cout<<"x+y="<<sum<<endl;

本行先输出字符串 “x+y=”, 然后输出 sum 的值。其中, “endl” 是输出操纵符, 其作用与 “\n” 相同, 表示本行结束换行。

第11行: return 0;

本行的作用是, 如果程序正常结束, 向操作系统返回一个数值 0

本程序的一次运行情况如下:

```
Please input two integers:
3✓
5✓
x+y= 8
```

从例 2.1 中可以看出, 第 6、7、8、10 行 4 个语句中的关键字 cin、cout 及运算符 “<<”、“>>” 在 C 语言中是没有的。它们正是 C++提供的新的输入输出方式。其中 cin 称为标准输入流对象, cout 是标准输出流对象, 它们都是 C++系统定义的对象。“>>” 是提取运算符 (也

称输入运算符),“<<”是插入运算符(也称输出运算符)。表达式

```
cout<<数据
```

表示把数据写到输出流对象 cout(可理解为屏幕)上。表达式

```
cin>>变量
```

表示从输入流对象输入 cin(可理解为键盘)读数据到变量中。

关于输入流对象和输出流对象的概念将在后面介绍,在此读者只要知道用“cin>>”和“cout<<”可以分别实现输入和输出就可以了。为了便于理解,我们把用 cin 和“>>”实现输入的语句简称为 cin 语句,把用 cout 和“<<”实现输出的语句简称为 cout 语句。

程序的第2行“#include <iostream>”是编译预处理命令,用来指示编译器在对程序进行预处理时,将文件 iostream 的代码嵌入到程序中该指令所在的地方。iostream 是 C++系统定义的一个头文件,在这个文件中声明了程序所需要的输入和输出操作的有关信息。流对象 cin、cout 及运算符“<<”、“>>”的定义,均包含在文件 iostream 中。由于这类文件常被嵌入在程序开始处,所以称之为头文件。

程序的第3行“using namespace std;”是针对命名空间 std 的指令,意思是使用命名空间 std。使用命名空间 std 可保证对 C++标准库操作的每一个特性都是唯一的,不至于发生命名冲突。关于命名空间的概念,本书将在第10章进行介绍,现在读者只需知道:使用“#include <iostream>”命令的同时,必须加上“using namespace std;”,否则编译时将出错。

由于 C++是从 C 语言发展而来的,为了与 C 语言兼容,C++保留了 C 语言中的一些规定。例如,在 C 语言中头文件用.h 作为后缀,如 stdio.h、math.h 等。为了与 C 语言兼容,许多 C++早期版本(如 VC++4.1 以前的版本)的编译系统头文件都是“*.h”形式,如 iostream.h 等。但后来 ANSI C++建议头文件不带后缀.h。近年推出的 C++编译系统新版本则采用了 C++的新方法,头文件名不再有“.h”扩展名,如采用 iostream、cmath 等。但为了使原来编写的 C++程序能够运行,在程序中,既可以选择使用旧版本的带后缀.h 的头文件,也可以使用新的不带后缀.h 的头文件。因此,例 2.1 也可以写成以下形式:

```
//sum.cpp
#include<iostream.h>           //带后缀.h的头文件
int main()                     //主函数首部
{ int x, y, sum;               //定义3个整型变量
  cout <<"Please input two integers:"<<"\n";
                                //提示用户由键盘输入两个整数
  cin>>x;                       //从键盘输入变量x的值
  cin>>y;                       //从键盘输入变量y的值
  sum=x+y;                    //将x+y的值赋给整型变量sum
  cout<<"x+y="<<sum<<endl;    //输出两个整数的和sum
  return 0;                   //如程序正常结束,向操作系统返回一个数值0
}
```

由于在这个程序中采用了带后缀.h 的头文件,这时就不需要再用“using namespace std;”声明了。

虽然两种头文件的说明方法同时并存,但是一定要注意,两种头文件不能混用。比如,若已经包含头文件 iostream,那么就不能再包含一个 math.h,而要代之以新的头文件 cmath。

2.2.2 C++程序的结构特性

例 2.1 是 C++ 的一个简单程序，可以使读者对 C++ 程序的格式有一个初步的了解。但是严格地说，例 2.1 并没有真正体现出 C++ 面向对象程序的风格。一个面向对象的 C++ 程序一般由类的声明和类的使用两大部分组成。即

面向对象程序 { 类的声明部分
类的使用部分

类的使用部分一般由主函数及有关子函数组成。例 2.2 就是一个典型 C++ 程序的框架结构，引入本例的目的是使读者对 C++ 面向对象程序的基本框架有一个初步的印象，我们将在后续的章节中作详细介绍。

例 2.2 C++ 程序的结构特性示例。

```
#include<iostream>           // 编译预处理命令
using namespace std;         //使用命名空间 std
class A{                      //声明一个类，类名为 A
    int x, y, z;              //声明类 A 的数据成员
    ...
    fun(){...}                //声明类 A 的成员函数 fun
    ...
};
int main()
{
    A a;                      //定义类 A 的一个对象 a
    ...
    a.fun();                  //调用对象 a 的成员函数 fun
    return 0;
}
```

类的声明部分

类的使用部分

在 C++ 程序中，程序设计始终围绕“类”展开。通过声明类，构建了程序所要完成的功能，体现了面向对象程序设计思想。在例 2.2 中首先声明了类 A，然后在主函数中创建了类 A 的对象 a，通过向对象 a 发送消息，调用成员函数 fun，完成了所需要的操作。

2.3 C++程序的编辑、编译、连接和运行

开发一个 C++ 程序的过程通常包括编辑、编译、连接、运行、调试等步骤。目前有许多软件产品可以帮助我们完成 C++ 程序的开发。例如，在 Windows 平台下有 Microsoft 公司的 Visual C++ 和 Borland 公司的 C++ Builder，在 Linux 平台下有 GUN 的 gcc 和 gdb。读者可以使用不同的 C++ 编译系统，在不同的环境下编译和运行一个 C++ 程序。

自从 Microsoft 公司发布 Visual C++ 以来，Visual C++ 已经成为 Windows 操作系统环境下最主要的应用系统开发工具之一，是目前用得最多的 C++ 编译系统，现在常用的是 Visual C++ 6.0 版本。

如果你使用的计算机上已经安装了 Visual C++ 6.0, 使用时只需单击 Windows 操作系统桌面上的“开始”→“程序”→“Microsoft Visual Studio 6.0”→“Visual C++ 6.0”命令, 就会出现如图 2-1 所示的 Visual C++ 6.0 的主窗口。

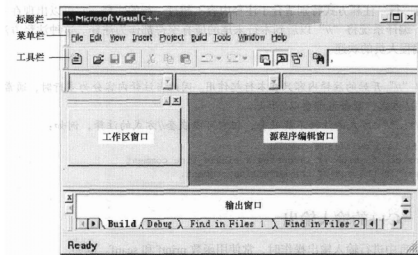


图 2-1 Visual C++ 6.0 的主窗口

主窗口由标题栏、菜单栏、工具栏、工作区窗口、源程序编辑窗口和输出窗口组成。Visual C++ 6.0 等 C++ 开发环境都带有 C 和 C++ 两种编译器, 当源程序文件扩展名为 .c 时, 启动 C 编译器, 当源程序文件扩展名为 .cpp 时, 启动 C++ 编译器。

C++ 程序的编辑、编译、连接和运行的方法和过程与 C 语言基本一样, 学过 C 语言上机操作的读者几乎不需要专门学习就可以完成 C++ 的上机操作过程。但需要注意的是, C 源程序文件扩展名为 .c, 而 C++ 源程序文件扩展名为 .cpp。

在本书的附录中将介绍 Visual C++ 6.0 上机实验的环境和运行程序的方法, 读者可以参阅。

2.4 C++对 C 的扩充

C++ 是从 C 语言发展而来, C 程序中的表达式、语句、函数和程序的组织方法等在 C++ 中仍可以使用。C++ 对 C 语言注入了面向对象的新概念, 同时也增加了一些非面向对象的新特性, 这些新特性使 C++ 程序比 C 程序更简洁或更安全。本章介绍 C++ 对 C 语言的非面向对象特性的扩充, 从第 3 章开始介绍 C++ 在面向对象方面的一些功能。

2.4.1 注释

C 语言提供了“块”注释方法。块注释用 “/*” 及 “*/” 作为注释分界符号, 例如:

```
/* This is the first line.
   This is the second line.
   This is the third line. */
```

C++除保留了这种注释方式外,还提供了一种更有效的“行”注释方式,该注释以“//”开始,到行尾结束。例如:

```
x=y+z; //This is a comment
```

C++的“行”注释方式特别适合于注释内容不超过一行的注释,它可以出现在一行中的语句之后,编译系统将“//”以后到本行末尾的所有字符都作为注释。这种注释方法灵活方便,很受编程人员的欢迎。

说明:

(1) 以“//”开始的注释内容只在本行起作用。因此当注释内容分为多行时,通常用/*...*/方式;如果用//方式,则每行都要以//开头。

(2) “/*...*/”方式的注释不能嵌套,但它可以嵌套//方式的注释,例如:

```
/* This is a multiline comment.
   inside of which // is nested a single_line comment
   Here is the end of the multiline comment.
*/
```

2.4.2 C++的输入输出

在C语言中进行输入输出操作时,常使用函数printf和scanf。例如:

```
int i;
float f;
...
scanf("%d", &i);
printf("%f", f);
```

C++除了可以照常使用这两个函数进行输入输出外,还增加了标准输入流对象cin和标准输出流对象cout来进行输入和输出。使用cin和cout进行输入输出更安全、更方便,上面的程序段可以写为:

```
int i;
float f;
...
cin>>i;
cout<< f;
```

1. 标准输入流对象cin

cin是标准的输入流对象,在程序中用于代表标准输入设备,通常指键盘。运算符“>>”在C++中仍保持C语言中的“右移”功能,但用于输入时扩充了其功能,表示将从标准输入流对象cin(通常指键盘)读取的数值传送给右方指定的变量。cin必须与输入运算符“>>”配套使用,请看下面的语句:

```
cin>>x;
```

此时,用户从键盘输入的数值会自动地转换为变量x的类型,并存入变量x内。x必须是基本数据类型,而不能是void类型。

运算符“>>”允许用户连续输入一连串数据,例如:

```
cin>>a>>b>>c;
```

它会按书写顺序从键盘上提取所要求的数据，并存入对应的变量中。两个数据间用空白符（空格、回车或 Tab 键）分隔。

说明：

(1) 在默认情况下，运算符“>>”将跳过空白符，然后读入后面的与变量类型相对应的值。因此，给一组变量输入值时可用空格或换行符将键入的数据间隔开。例如：

```
int i;
float x;
cin>>i>>x;
```

在输入时，可以采用下面形式：

```
23 56.78
```

或

```
23
56.78
```

(2) 当输入字符串（即类型为 char* 的变量）时，提取运算符“>>”的作用是跳过空白，读入后面的非空白字符，直到遇到另一个空白字符为止，并在串尾放一个字符串结束标志‘\0’。因此，输入字符串遇到空格时，就当作本数据输入结束。例如：

```
char* str;
cin>>str;
```

当键入的字符串为

```
Object_Oriented Programming!
```

则输入后，str 中的字符串是“Object_Oriented”，而后面的字符串“Programming!”被略去了。

(3) 数据输入时，系统除检查是否有空白外，还检查输入数据与变量的匹配情况。例如，对于语句

```
cin>>i>>x;    //i 为 int 型，x 为 float 型
```

若从键盘输入：

```
56.79 32.5
```

得到的结果就不是预想的

```
i=56, x=32.5,
```

而是

```
i=56, x=0.79,
```

这是因为，系统是根据变量的类型来分隔输入的数据的。在这种情况下，系统把 56.79 中小数点前面的整数部分赋给了整型变量 i，而把剩下的 0.79 赋给了浮点型变量 x。

2. 标准输出流对象 cout

cout 是标准输出流对象，在程序中用于代表标准输出设备，通常指屏幕。运算符“<<”在 C++ 中仍保持 C 语言中的“左移”操作，但用于输出时扩充了其功能，表示将右方变量的值写到标准输出流 cout 对象中，即显示在屏幕上。cout 必须与输出运算符“<<”配套使用，

如执行下面的语句后：

```
cout<<y;
```

变量 y 的值将显示在屏幕上。

使用插入运算符“<<”进行输出操作时，可以把多个不同类型的数据组合在一条语句中，也可以输出表达式的值，使用起来很方便。例如：

```
cout <<a+b<<c;
```

它会按书写顺序将“ $a+b$ ”和 c 的值输出到屏幕上。

```
int n=456;
```

```
double d=3.1416;
```

```
cout<<"n="<<n<<"，d="<<d<<'\\n';
```

就是由整型和字符串型数据组合在一起的语句。编译程序根据出现在“<<”操作符右边的变量或常量的类型来决定调用形参为哪种标准类型的运算符重载函数。

上述语句的输出结果为

```
n=456, d=3.1416
```

说明：

(1) 使用 `cin` 或 `cout` 进行 I/O 操作时，在程序中必须嵌入头文件 `iostream`，否则编译时要产生错误。

(2) 在 C++ 程序中，我们仍然可以沿用传统的 `stdio` 函数库中的 I/O 函数，如 `printf` 函数、`scanf` 函数或其他 C 输入输出函数，但只有使用“`cin>>`”和“`cout<<`”才能显示 C++ 的输入和输出风格。输入或输出时，`cin` 和 `>>`、`cout` 和 `<<` 要配套使用。

(3) 在 C 语言中，常用‘`\n`’实现换行，而在 C++ 中增加了换行操纵符 `endl`，其作用与‘`\n`’一样。例如，以下两个语句的操作是等价的：

```
cout<<"x="<<x<<endl;
```

```
cout<<"x="<<x<<'\\n';
```

(4) 前面用 `cout` 和 `cin` 输出输入数据时，全部使用了系统默认的格式。实际上，我们也可以对输入和输出的格式进行控制。例如，我们可用设置域宽的操纵符 `setw(n)`，来控制输出数据的宽度。请看下面的例子。

例 2.3 设置域宽的操纵符 `setw(n)` 的使用。

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ cout<<123<<endl;           //①
  cout<<setw(6)<<456<<endl;    //②
  return 0;
}
```

程序运行结果为：

```
123
```

```
456
```

下面分析输出结果。

第①条 `cout` 语句按默认方式输出 123, 即 123 占域宽为 3。

第②条 `cout` 语句首先用操纵符 `setw(6)` 设置域宽为 6, 之后按域宽 6 输出 456, 即 456 占域宽为 6。

由于操纵符 `setw(n)` 在头文件 `iomanip` 中定义, 所以需要在程序的开头加上编译预处理命令 `#include<iomanip>`”。有关 C++ 输入输出的格式控制的方法将在第 9 章中详细介绍。

2.4.3 灵活的局部变量说明

在 C 语言程序中, 全局变量声明必须在任何函数之前, 局部变量必须集中在可执行语句之前。而 C++ 的变量声明非常灵活, 它允许变量声明与可执行语句在程序中交替出现。这样, 程序员就可以在使用一个变量时才声明它。例如, 在 C 中, 下面的函数 `f` 是不正确的:

```
f()
{ int i;
  i=10;
  int j;          // 在 C 语言中, 这条语句是不允许的, 但在 C++ 中是正确的
  j=25;
  ...
}
```

由于在函数 `f` 中可执行语句 `“i=10;”` 插在两个变量说明之间, 在 C 语言环境编译时将指示有错, 并中止对函数 `f` 的编译。但在 C++ 中, 以上程序段是正确的, 编译时不会出错。

C++ 允许在代码块中的任何地方说明局部变量, 它所说明的变量从其说明点到该变量所在的最小分程序末的范围内有效。需要强调的是, 局部变量的说明一定要符合“先定义、后使用”的规定。

2.4.4 `const` 修饰符

在 C 语言中, 习惯使用 `#define` 来定义常量, 例如:

```
#define LIMIT 100
```

实际上, 这种方法只是在预编译时进行字符置换, 把程序中出现的标识符 `LIMIT` 全部置换为 100。在预编译之后, 程序中不再有 `LIMIT` 这个标识符。`LIMIT` 不是变量, 没有类型, 不占用存储单元, 而且容易出错。

C++ 提供了一种更灵活、更安全的方式来定义常量, 即使用 `const` 修饰符来定义常量, 例如:

```
const int LIMIT=100;
```

这个常量 `LIMIT` 是有类型的, 占用存储单元, 有地址, 可以用指针指向它。常量一旦被建立, 在程序的任何地方都不能再更改它。`const` 方便实用, 避免了用 `#define` 定义常量时出现的缺点。因此 C++ 建议用 `const` 取代 `#define` 定义常量。

`const` 既可以放在常量的类型修饰符前, 也可以放在类型修饰符后。例如:

```
const double pi=3.14;
```

和

```
double const pi=3.14;
```

是等价的。

`const` 可以与指针结合使用, 此时一定要注意其位置, 如下面两条语句的功能是完全不同的:

(1) `const char* pc="abcd";`

这个语句的含义为: 声明一个名 `pc` 的指针变量, 它指向一个字符型常量, 初始化 `pc` 为指向字符串 "abcd"。

由于使用了 `const`, 不允许改变指针所指的常量, 因此以下语句是错误的:

```
pc[3]='x';
```

但是, 由于 `pc` 是一个指向常量的普通指针变量, 不是常指针, 因此可以改变 `pc` 所指的地址。例如下列语句是允许的:

```
pc="efgh";
```

该语句赋给了指针另一个字符串的地址, 即改变了 `pc` 的值。

(2) `char* const pc="abcd";`

这个语句的含义为: 声明一个名为 `pc` 的指针变量, 该指针是指向字符型数据的常指针, 用 "abcd" 的地址初始化该常指针。

创建一个常指针, 就是创建一个不能移动的固定指针, 即不能改变指针所指的地址, 但是它所指的地址中的数据可以改变。例如:

```
pc[3]='x';           // 合法, 可以改变常指针 pc 所指地址中的数据
pc="efgh";           // 出错, 不能改变常指针所指的地址
```

第 1 个语句改变了常指针所指地址中的数据, 这是允许的; 但第 2 个语句要改变常指针所指的地址, 这是不允许的。

说明:

(1) `#define` 可以看成是一个程序预处理语句, 只能用于在程序的开头位置定义全局的常量; 而 `const` 可以在程序中的任意位置定义常量, 所定义常量的作用域亦随定义位置而变化。

(2) 如果用 `const` 定义的是一个整型常量, 关键字 `int` 可以省略。所以下面的两行定义是等价的:

```
const int LIMIT=100;
const LIMIT=100;
```

(3) 与 `#define` 定义的常量有所不同, `const` 定义的常量可以有自己的数据类型, 这样 C++ 的编译程序可以进行更加严格的类型检查, 具有良好的编译时的检测性。

2.4.5 函数原型

在 C 语言程序中, 如果函数调用的位置在函数定义之前, 则 C 语言建议, 在函数调用之前, 采用函数原型声明的形式, 对所调用的函数进行声明。请看下面的例子。

例 2.4 在 C 语言程序中函数原型的声明。

```
#include<stdio.h>
int add(int a, int b);           /* 函数原型声明 */
int main()                       /* 主函数 */
{ int x, y, sum;                 /* 定义 3 个整型变量 */
```

```

printf("Enter two numbers:\n");    /* 提示用户输入两个数的值 */
scanf("%d", &x);                  /* 从键盘输入变量 x 的值 */
scanf("%d", &y);                  /* 从键盘输入变量 y 的值 */
sum=add(x, y);                    /* 调用函数 add, 将得到的值赋给变量 sum */
printf("x+y= %d ", sum);          /* 输出两个数的和 sum 的值 */
return 0;
}

int add(int a, int b)              /* 定义 add 函数, 函数的返回值类型为整型 */
{ int c;                          /* 定义一个整型变量 */
  c=a+b;                          /* 计算两个数的和 */
  return c;                       /* 将 c 的值返回, 通过 add 带回调用处 */
}

```

在本例中采用了函数原型对函数 add 进行声明。但这并不是强制性的, 在编译时并不严格要求。在 C 语言中声明函数原型时, 也可以采用简化的声明形式, 如下面几种声明的形式都是合法的, 都能通过编译。

```

int add(int a, int b);            /* add 函数原型声明 */
int add();                        /* 可以不列出 add 函数的参数表 */
add();                            /* 当 add 函数的返回类型是整型时, 可以省略 int */

```

在 C++ 中, 如果函数调用的位置在函数定义之前, 则要求在函数调用之前必须对所调用的函数作函数原型声明, 以说明函数的名称、参数类型与个数, 以及函数返回值的类型。其主要目的是让 C++ 编译程序进行检查以确定调用函数的参数以及返回值类型与事先定义的原型是否相符, 以保证程序的正确性。例如:

```
int add(int a, int b);
```

就是函数 add 的原型。

函数原型的语法形式一般为:

返回值类型 函数名 (参数表);

函数原型是一条语句, 它必须以分号结束。它由函数的返回值类型、函数名和参数表构成。参数表包含所有参数及它们的类型, 参数之间用逗号分开。下面是用 C++ 将例 2.4 改写的例子。

例 2.5 在 C++ 程序中函数原型的声明。

```

#include<iostream>
using namespace std;
int add(int a, int b);            //函数原型声明
int main()                       //主函数
{ int x, y, sum;                 //定义 3 个整型变量
  cout<<"Enter two numbers:\n"; //提示用户输入两个数的值
  cin>>x;                        //从键盘输入变量 x 的值
  cin>>y;                        //从键盘输入变量 y 的值
  sum=add(x, y);                 //调用函数 add, 将得到的值赋给变量 sum
  cout<<"x+y= "<<sum;           //输出两个数的和 sum 的值
  return 0;
}

```

```

int add(int a, int b)           // 定义 add 函数，函数的返回值类型为整型
{ int c;                       // 定义一个整型变量
  c=a+b;                       // 计算两个数的和
  return c;                    // 将 c 的值返回，通过 add 带回调用处
}

```

在程序中，当一个函数的定义在后，而对它的调用在前时，必须将该函数原型声明放在调用语句之前；但当一个函数的定义在前，而对它的调用在后时，一般就不必再单独给出它的原型声明了。因为，这时函数定义的说明部分起到了函数原型声明的作用。

例 2.6 函数定义在前，调用在后的示例。

```

#include<iostream>
using namespace std;
int add(int a, int b)           // 函数 add 定义在前，而对它的调用在后
{ int c;
  c=a+b;
  return c;
}

int main()                     // 主函数
{ int x, y, sum;
  cout<<"Enter two numbers:\n";
  cin>>x;
  cin>>y;
  sum=add(x, y);               //调用函数 add
  cout<<"x+y= "<<sum;
  return 0;
}

```

说明：

(1) 函数原型的参数表中可不包含参数的名字，而只包含它们的类型。例如，以下的函数原型是完全合法的：

```
long Area(int, int);
```

该原型声明一个返回值类型为 long、有两个 int 型参数、函数名为 Area 的函数。尽管这一结构是合法的，但是加上参数名将使原型更加清楚。例如，带有参数的同一函数原型可以书写成：

```
long Area(int length, int width);
```

这样，可以很清楚地看出，第 1 个参数表示长度，第 2 个参数表示宽度。

(2) 函数定义由函数说明和函数体两个部分构成。函数说明部分与函数原型基本一样，但函数说明部分中的参数必须给出参数的名字，而且不能包含结尾的分号，例如：

```

long Area(int length, int width)    //函数的说明部分
{ ...
  return(length*width);
}

```

(3) 主函数 main 不必进行原型说明，因为它被看成一个自动说明原型的函数。主函数是第 1 个被执行的函数，而且不存在被别的函数调用的问题。

(4) 原型说明中没有指出返回值类型的函数（包括主函数 main），C++默认该函数的返

回值类型是 int。因此以下的原型说明在 C++ 中是等价的。

```
cal(float a, int c);           // 默认的返回值类型是 int 型
int cal(float a, int c);       // 指明返回值类型是 int 型
```

标准 C++ 要求 main 函数必须声明为 int 型, 即要求主函数带回一个整型返回值。C++ 通常是这样处理的: 如果程序正常结束, 则在 main 函数的最后加一条语句 “return 0;”, 向操作系统返回数值 0。如果函数执行不正常, 则返回数值 -1。

(5) 如果一个函数没有返回值, 则必须在函数原型中注明返回类型为 void, 这时函数的最后就不必有 “return ;” 之类的返回语句了。需要说明的是: 标准 C++ 要求 main 函数声明为 int 型, 但是目前使用的 C++ 编译系统并未完全执行 C++ 的这一规定, 如果主函数首行写成 “void main()” 也能通过编译。我们建议读者执行标准 C++ 的这一规定, 编写程序时注意在 main 前面加 int, 同时在 main 函数的最后加上一条 “return 0;” 语句。

2.4.6 内联函数

在函数说明前冠以关键字 “inline”, 该函数就被声明为内联函数, 又称内置函数。每当程序中出现对该函数的调用时, C++ 编译器使用函数体中的代码插入到调用该函数的语句之处, 同时用实参取代形参, 以便在程序运行时不再进行函数调用。

指定内联函数的方法很简单, 只须在函数首行的左端加上一个关键字 inline 即可。定义内联函数的一般格式为:

```
inline 返回值类型 函数名(参数表)
{
    函数体
}
```

下面的程序定义了一个内联函数。

例 2.7 将函数指定为内联函数。

```
#include <iostream>
using namespace std;
inline int box (int i, int j, int k)    //定义 box 为内联函数, 注意左端有 inline
{ return i*j*k;
}
int main()
{ int a, b, c, v;
  cin>>a>>b>>c;
  v=box(a, b, c);
  cout<<"a*b*c= "<<v<<endl;
  return 0;
}
```

程序运行情况如下:

```
345/      (输入 3、4、5 分别给 a、b、c)
a*b*c= 60
```

由于在定义函数 box 时指定它为内联函数, 因此编译系统在遇到函数调用 box(a, b, c) 时, 就用 box 函数体的代码代替 box(a, b, c), 同时将实参代替形参。这样, “v=box(a, b,

c)”就被置换成：

```
{ int i=a;
  int j=b;
  int k=c;
  v=i*j*k;
}
```

为什么要引入内联函数呢？这主要是为了消除函数调用时的系统开销，以提高运行速度。我们知道，在程序执行过程中调用函数时，系统要将程序当前的一些状态信息（例如现场和返回地址等）存到栈中，同时转到函数的代码处去执行函数体语句，这些参数保存与传递的过程中需要时间和空间的开销，使得程序执行效率降低，特别是在程序频繁地调用函数时，这个问题会变得更为严重。

说明：

（1）内联函数在第1次被调用之前必须进行完整的定义，否则编译器将无法知道应该插入什么代码。

（2）在内联函数体内一般不能含有复杂的控制语句，如 for 语句和 switch 语句等。

（3）使用内联函数是一种用空间换时间的措施，若内联函数较长，且调用太频繁时，程序将加长很多。如果将一个复杂的函数定义为内联函数，反而会使程序代码加长很多，增大开销。在这种情况下，多数编译器会自动将其转换为普通函数来处理。通常只有规模很小（一般为1~5条语句）而使用频繁的函数才定义为内联函数，这样可大大提高运行速度。

2.4.7 带有默认参数的函数

一般情况下，实参个数应与形参个数相同，但 C++ 允许实参个数与形参个数不同。方法是在说明函数原型时（若没有说明函数的原型，则应在函数定义时），为一个或多个形参指定默认值，以后调用此函数时，若省略其中某一实参，C++ 自动地以默认值作为相应参数的值。例如有一个求矩形面积的函数，其函数原型说明为：

```
int area(int x=5, int y=10);
```

则 x 边与 y 边的默认参数值分别为 5 与 10。

当进行函数调用时，编译系统按从左向右顺序将实参与形参结合，若未指定足够的实参，则编译系统按顺序用函数原型中的形参默认值来补足所缺少的实参。例如，以下的函数调用都是允许的：

```
area(100, 50 )    //x=100, y=50, 计算 x 和 y 分别为 100 和 50 的矩形面积
area(25)          //相当于 area(25, 10), 结果为 x=25, y=10
area( )           //相当于 area(5, 10), 结果为 x=5, y=10
```

可以看到，在调用带有默认参数的函数时，实参的个数可以与形参不同，实参未给定的，可从形参的默认值得到值。利用这一特性，可以使函数的使用更加灵活。

说明：

（1）在声明函数时，所有指定默认值的参数都必须出现在不指定默认值的参数的右边。因为实参与形参的结合是从左至右的顺序进行的，第1个实参必然与第1个形参结合，第2个实参必然与第2个形参结合……，因此指定默认值的参数必须放在形参表列中的最右端，

否则出错。例如：

```
int fun(int i, int j=5, int k);
```

是错误的，因为在指定默认参数的 `int j=5` 后，不应再说明不带默认参数的 `int k`。若改为：

```
int fun(int i, int k, int j=5);
```

则是正确的。

(2) 在函数调用时，若某个参数省略，则其后的参数皆应省略而采用默认值。不允许某个参数省略后，再给其后的参数指定参数值。例如，不允许出现以下调用：

```
area(, 20)
```

(3) 如果函数的定义在函数调用之前，则应在函数定义中指定默认值。如果函数的定义在函数调用之后，则函数调用之前需要有函数声明，此时必须在函数声明中给出默认值，在函数定义时就不要给出默认值了(因为如果在函数声明与函数定义时都给出默认值，有的 C++ 编译系统会给出“重复指定默认值”的报错信息)。

2.4.8 函数的重载

在传统的 C 语言中，在同一作用域内，函数名必须是唯一的，也就是说不允许出现同名的函数。例如，当要求编写求整数、长整型数和双精度数的二次方的函数时，若用 C 语言来处理，必须编写 3 个函数，这 3 个函数的函数名不允许同名。例如：

```
Isquare(int i);           //求整数的二次方
Lsquare(long l);         //求长整型数的二次方
Dsquare(double d);       //求双精度数的二次方
```

当使用这些函数求某个数的二次方时，必须调用合适的函数，也就是说，用户必须记住 3 个函数，虽然这 3 个函数的功能是相同的。

在 C++ 中，函数可以重载。这意味着，只要函数参数的类型不同，或者参数的个数不同，或者二者兼而有之，两个或者两个以上的函数可以使用相同的函数名。在同一作用域内，当两个或者两个以上的函数共用一个函数名时，称为函数的重载。被重载的函数称为重载函数。

由于 C++ 支持函数重载，上面 3 个求二次方的函数可以起一个共同的名字 `square`，但它们的参数类型仍保留不同。当用户调用这些函数时，编译系统就会根据实参的类型来确定调用哪个重载函数。因此，用户调用求二次方的函数时，只需记住一个 `square` 函数，至于调用哪一个重载函数由编译系统来完成。上述例子我们可以用下面的程序来实现。

例 2.8 参数类型不同的函数重载。

```
#include<iostream>
using namespace std;
int square(int i)
{ return i*i;
}
long square(long l)
{ return l*l;
}
double square(double d)
{ return d*d;
}
```

```
int main()
{ int i=12;
  long l=1234;
  double d=5.67;
  cout<<i<<' '*<<i<<'='<<square(i)<<endl;
  cout<<l<<' '*<<l<<'='<<square(l)<<endl;
  cout<<d<<' '*<<d<<'='<<square(d)<<endl;
  return 0;
}
```

程序运行结果如下:

```
12*12=144
1234*1234=1522756
5.67*5.67=32.1489
```

在 main 函数中 3 次调用了 square 函数, 实际上是调用了 3 个不同的函数版本。由系统根据传送的不同参数类型来决定调用哪个函数版本。例如, 使用 square(i) 来调用函数, 因为 i 为整型变量, 所以 C++ 系统将调用求整数二次方的函数版本。可见, 利用重载概念, 用户在调用函数时非常方便。

由函数重载的定义可以看出, 函数重载的条件是:

- (1) 函数名相同;
- (2) 函数参数的特征(形参类型或者个数)不同。

编译时, 系统会根据实参的个数和类型, 决定调用哪个函数版本。

下面是两个参数个数不同的函数重载的例子。

例 2.9 参数个数不同的函数重载。

```
#include<iostream>
using namespace std;
int mul(int x, int y)
{ return x*y;
}
int mul(int x, int y, int z)
{ return x*y*z;
}
int main()
{ int a=3, b=4, c=5;
  cout<<a<<' '*<<b<<'='<<mul(a, b)<<endl;
  cout<<a<<' '*<<b<<' '*<<c<<'='<<mul(a, b, c)<<endl;
  return 0;
}
```

程序运行结果如下:

```
3*4=12
3*4*5=60
```

例中的函数 mul 被重载, 这两个重载函数的参数个数是不同的。编译程序根据传送参数的数目决定调用哪一个函数。

说明:

- (1) 调用重载函数时, 函数返回值类型不在参数匹配检查之列。因此, 若两个函数的参

数个数和类型都相同，而只有返回值类型不同，则不允许重载。例如：

```
int Add(int x, int y){ return x+y; }
double Add(int x, int y) { return x+y; }
```

虽然这两个函数的返回值类型不同。但是由于参数个数和类型完全相同。因此C++编译系统无法从函数的调用形式上判断哪一个函数与之匹配。当执行以下的函数调用时：

```
cout<<Add(10, 20)<<endl;
```

编译时将出现出错信息：“overloaded function differs only by return type”。

(2) 函数的重载与带默认值的函数一起使用时，有可能引起二义性。例如，有以下两个函数：

```
int fun(int x=0, int y=0){return x+y;}
int fun(int r){return r;;}
```

当执行以下的函数调用时：

```
cout<<fun(20)<<endl;;
```

编译时将出现出错信息：“ambiguous call to overloaded function”。这是因为编译系统无法确定调用哪一个函数。

(3) 在函数调用时，如果给出的实参和形参类型不相符，C++编译器会自动执行类型转换工作。如果转换成功，则程序继续执行，在这种情况下，有可能产生不可识别的错误。

例如，有两个函数的原型如下：

```
int Add(int x, int y){ return x+y;}
long Add(long x, long y){ return x+y;}
```

虽然这两个函数满足函数重载的条件，但是，如果我们用下面的数据去调用，就会出现不可分辨的错误：

```
cout<<Add(10.1, 20.2)<<endl;
```

编译时将出现出错信息：“ambiguous call to overloaded function”。这是因为编译器无法确定将10.1和20.2转换成int还是long类型的原因造成的。

2.4.9 作用域运算符“::”

通常情况下，如果有两个同名变量，一个是全局的，另一个是局部的，那么局部变量在其作用域内具有较高的优先权，它将屏蔽全局变量。

下面的例子说明了这一点。

例 2.10 全局变量和局部变量同名。

```
#include<iostream>
using namespace std;
int avar=10; //全局变量 avar
int main()
{ int avar; //局部变量 avar
  avar=25;
  cout<<"avar is"<<avar<<endl; //输出局部变量 avar 的值
  return 0;
}
```

程序执行结果如下：

```
avar is 25
```

此时，在 main 函数的输出语句中，使用的变量 avar 是 main 函数内定义的局部变量，因此打印的是局部变量 avar 的值。

如果希望在局部变量的作用域内使用同名的全局变量，可以在该变量前加上 "::"，此时::avar 代表全局变量 avar，“::”称为作用域运算符。

请看下面的例子。

例 2.11 作用域运算符的使用。

```
#include<iostream>
using namespace std;
int avar;
int main()
{ int avar;
  avar=25; //给局部变量 avar 赋值
  ::avar=10; //给全局变量 avar 赋值
  cout<<"local avar ="<<avar<<endl; //输出局部变量 avar 的值
  cout<<"global avar ="<<::avar<<endl; //输出全局变量 avar 的值
  return 0;
}
```

程序运行结果如下：

```
localavar=25
globalavar=10
```

从这个例子可以看出，作用域运算符可用来解决局部变量与全局变量的重名问题。即在局部变量的作用域内，可用 "::" 对被屏蔽的同名全局变量进行访问。

2.4.10 强制类型转换

在 C 语言表达式中不同类型的数据会自动地转换类型。有时，编程者还可以利用强制类型转换将不同类型的数据进行转换。例如，要把一个整数(int)转换为双精度型数(double)，可使用如下的格式：

```
int i=10;
double x=(double)i;
```

C++支持这样的格式，但还提供了一种更为方便的类似于函数调用的格式，使得类型转换的执行看起来好像调用了一个函数。上面的语句可改写成：

```
int i=10;
double x=double(i);
```

以上两种方法 C++都能接受，但推荐使用后一种方式。

2.4.11 运算符 new 和 delete

C 语言使用函数 malloc 和 free 动态分配内存和释放动态分配的内存，然而 C++使用运算符 new 和 delete 能更好、更简单地进行内存的分配和释放。但是，为了与 C 语言兼容，C++中仍保留了 malloc 和 free 这两个函数。

运算符 `new` 用于内存分配的最基本形式为:

指针变量名 = new 类型;

在程序运行过程中, 运算符 `new` 就从堆的一块自由存储区中为程序分配一块与类型字节数相适应的内存空间, 并将该块内存的首地址存于指针变量中。例如:

```
int *pi;           //定义一个整型指针变量 pi
pi=new int;        //new 动态分配存放一个整数的内存空间, 并将其首地址赋给指针变量 pi
char *pc;          //定义一个字符型指针变量 pc
pc=new char;       //new 动态分配存放一个字符的内存空间, 并将其首地址赋给指针变量 pc
double *pd;        //定义一个双精度型指针变量 pd
pd=new double;     //new 动态分配存放一个双精度数的内存空间,
                  //并将其首地址赋给指针变量 pd
```

运算符 `delete` 用于释放运算符 `new` 分配的存储空间。该运算符释放存储空间的基本形式为:

delete 指针变量名;

其中, 指针变量保存着 `new` 分配的内存的首地址。例如:

```
delete pi;         //将上述 new 动态分配的存放整数的内存空间释放
delete pc;         //将上述 new 动态分配的存放字符的内存空间释放
delete pd;         //将上述 new 动态分配的存放双精度型数的内存空间释放
```

下面是使用 `new` 和 `delete` 的一个简单例子。

例 2.12 使用 `new` 和 `delete` 的简单例子。

```
#include<iostream>
using namespace std;
int main()
{ int *ptr;          //定义一个整型指针变量 ptr
  ptr=new int;        //动态分配一个整型存储空间, 并将首地址赋给 ptr
  *ptr=10;
  cout<<*ptr;
  delete ptr;         //释放指针 ptr 指向的存储空间
  return 0;
}
```

程序执行结果如下:

10

该程序定义了一个整型指针变量 `ptr`, 用 `new` 分配了一块存放一个整数的内存空间, 并将首地址赋给指针 `ptr`, 然后在这内存块中赋予初值 10, 并将其打印出来, 最后释放 `ptr` 指向的存储空间。

虽然 `new` 和 `delete` 完成的功能类似于函数 `malloc` 和 `free`, 但是它们有以下几个优点:

(1) `new` 可以根据数据类型自动计算所要分配内存的大小, 而使用 `malloc` 函数时必须使用 `sizeof` 函数来计算所需要的字节数, 这就减少了发生错误的可能性。

(2) `new` 能够自动返回正确的指针类型, 而 `malloc` 函数的返回值类型一律为 `void*`, 必须在程序中进行强制类型转换, 才能使其指针指向具体的数据。

下面我们对 `new` 和 `delete` 的使用再作几点说明。

(1) 使用 `new` 可以为数组动态分配内存空间, 这时需要在类型名后面缀上数组大小。

例如:

```
int *pi=new int[10];
```

这时 `new` 为具有 10 个数组元素的整型数组分配了内存空间, 并将其首地址赋给了指针 `pi`。

使用 `new` 为多维数组分配空间时, 必须提供所有维的大小, 例如:

```
int *pi=new int[2][3][4];
```

释放动态分配的数组存储区时, 可使用如下的 `delete` 格式:

delete []指针变量名;

在此指针变量名前只用一对方括号, 无须指出所删除数组的维数和大小。例如:

```
delete []pi; //在指针变量前面加一个方括号, 表示对数组空间的操作
```

(2) `new` 可在为简单变量分配内存的同时, 进行初始化。其基本形式为:

指针变量名 = new 类型(初值);

初始值放在“类型”后面的圆括号内。请看下面的例子。

例 2.13 为简单变量分配内存的同时, 进行初始化。

```
#include<iostream>
using namespace std;
int main()
{ int *p;
  p=new int(99); //动态分配内存空间, 并将 99 作为初始值赋给它
  cout<<"p";
  delete p;
  return 0;
}
```

但是, `new` 不能对动态分配的数组存储区进行初始化。

(3) 使用 `new` 动态分配内存时, 如果没有足够的内存满足分配要求, 则动态分配空间失败, 有些编译系统将返回空指针 `NULL`, 因此可以对内存的动态分配是否成功进行检查。请看以下例子。

例 2.14 对动态内存分配是否成功进行检查。

```
#include<iostream>
using namespace std;
int main()
{ int *p;
  p=new int;
  if(!p)
  { cout <<"allocation failure\n";
    return 1;
  }
  *p=20;
  cout <<"p";
  delete p;
  return 0;
}
```

若动态分配内存失败, 此程序将在屏幕上显示“allocation failure”。内存动态分配成功后

不宜变动指针的值，否则在释放存储空间时会引起系统内存管理失败。

(4) 用 new 分配的存储空间不会自动释放，只能通过 delete 释放。因此，要适时释放动态分配的存储空间。

2.4.12 引用

1. 引用的概念

引用 (reference) 是 C++ 对 C 语言的一个重要的扩充。在 C++ 中，变量的“引用”就是变量的别名。

建立引用的作用是为变量另起一个名字，以便在需要时可以方便、间接地引用该变量，这就是引用名称的由来。当声明了一个引用时，必须同时用另一个变量的名字来将它初始化，即声明它代表哪一个变量，是哪一个变量的别名。这样，对一个引用的所有操作，实际上都是对它所代表的变量的操作，就如同对一个人来说，即使有三四个名字，实际就是同一个人，用这三四个人名所做的事情，其实就是那一个人所做的事情。声明一个引用的格式如下：

类型 & 引用名 = 已定义的变量名;

例如：

```
int i=5;
int &j=i;           //声明 j 是一个整型变量 i 的引用，用整型变量 i 对它进行初始化
```

在此，j 是一个整型变量的引用，用整型变量 i 对它进行初始化，这时 j 就可看做是变量 i 的引用，即是变量 i 的别名。经过了这样声明后，使用 i 和 j 的作用相同，都代表同一个变量。上述声明中“&”是引用声明符，此时它不代表地址。

注意：不要把声明语句“int &j=i;”理解为“将变量 i 的值赋给引用 j”，它的作用是使 j 成为 i 的引用，即 i 的别名。

例 2.15 了解引用和变量的关系。

```
#include<iostream>
using namespace std;
int main()
{ int i=10;
  int &j=i;                               //声明 j 是一个整型变量 i 的引用
  i=30;                                   //变量 i 的值变化了
  cout<<"i="<<i<<" j="<<j<<"\n";       //引用 j 的值也随着变量 i 的值一起变化
  j=80;                                   //引用 j 的值变化了
  cout<<"i="<<i<<" j="<<j<<"\n";       //变量 i 的值也随着引用 j 的值一起变化
  return 0;
}
```

程序执行结果如下：

```
i=30 j=30
i=80 j=80
```

由运行结果可以看出，i 和 j 的值同步更新，当 i 变化时，j 也随之变化，反之亦然。引用与其所代表的变量共享同一内存单元，系统并不为引用另外分配存储空间。实际上，编译系统使引用和其代表的变量具有相同的地址。

例 2.16 引用和其代表的变量具有相同的地址。

```
#include<iostream>
using namespace std;
int main()
{ int i=10;
  int &j=i;                                //声明 j 是一个整型变量 i 的引用
  cout<<"i="<<i<<" j="<<j<<"\n";        //引用 j 的值随着变量 i 的值一起变化
  cout<<"变量 i 的地址:"<<&i<<"\n";      //输出变量 i 的地址
  cout<<"引用 j 的地址:"<<&j<<"\n";      //输出引用 j 的地址
  return 0;
}
```

程序执行结果如下：

```
i=10 j=10
变量 i 的地址:0012FF7C
引用 j 的地址:0012FF7C
```

由运行结果可以看出，i 和 j 的值相同，且使用内存的同一地址，此例中变量 i 和引用 j 的地址均为 0012FF7C（注意，此地址视实际运行而有所不同）。

说明：

（1）引用并不是一种独立的数据类型，它必须与某一种类型的变量相联系。在声明引用时，必须立即对它进行初始化，不能声明完成后再赋值。例如下述声明是错误的。

```
int i=10;
int &j;          //错误，没有指定 j 代表哪个变量
j=i;            //不能声明完成后再赋值
double a;
int &b=a;        //错误，声明 b 是一个整型变量的别名，而 a 不是整型变量
```

（2）为引用提供的初始值，可以是一个变量或另一个引用。例如：

```
int i=5;        //定义整型变量 i
int &j1=i;       //声明 j1 是整型变量 i 的引用(别名)
int &j2=j1;      //声明 j2 是整型引用 j1 的引用(别名)
```

这样定义后，变量 i 有两个别名，即 j1 和 j2。

（3）指针是通过地址间接访问某个变量，而引用是通过别名直接访问某个变量。每次使用引用时，可以不用像指针那样书写间接运算符“*”，因而使用引用比使用指针更直观、方便，直截了当，不必通过运算符“*”兜圈子，可以简化程序，便于理解。

请看以下的例子。

例 2.17 比较引用和指针的使用方法。

```
#include<iostream>
using namespace std;
int main()
{ int i=15;                                //定义整型变量 i，赋初值为 15
  int *iptr=&i;                            //定义指针变量 iptr，将变量 i 的地址赋给 iptr
  int &rptr=i;                            //声明变量 i 的引用 rptr，rptr 是变量 i 的别名
  cout<<"i is " <<i<<endl;              //输出变量 i 的值
```

```

cout<<"*iptr is "<<*iptr<<endl;    //通过指针变量 iptr, 输出变量 i 的值, 需要
                                   //通过运算符 "*"
cout<<"rptr is "<<rptr<<endl;      //通过引用 rptr, 输出变量 i 的值, 不需
                                   //要通过运算符 "*" , 方便、直观

return 0;
}

```

程序运行结果如下:

```

i is 15
*iptr is 15
rptr is 15

```

从这个程序可以看出, 如果要使用指针变量 `iptr` 所指的变量 `i`, 必须用 “*” 来间接引用指针; 而使用引用 `rptr` 所代表的变量 `i`, 不必书写间接引用运算符 “*”。

(4) 引用在初始化后不能再被重新声明为另一个变量的引用(别名)。例如:

```

int i, k;           //定义 i 和 k 是整型变量
int &j=i;           //声明 j 是整型变量 i 的引用(别名)
j=&k                //错误, 企图重新声明 j 是整型变量 k 的引用(别名)

```

2. 引用作为函数参数

C++ 提供引用, 其主要的用途就是将引用作为函数的参数。在讨论这个问题之前, 先回顾一下, 在 C 语言中, 传递函数参数的两种情况。

(1) 变量名作为函数参数。这时实参传给形参的是实参变量的值, 即“传值调用”。这种传递是单向的, 在执行函数期间形参值发生的变化并不传回给实参, 因为在调用函数时, 形参和实参不是占有同一个存储单元。下面的程序无法实现两个变量值的互换。

例 2.18 变量名作为函数参数。

```

#include<iostream>
using namespace std;
void swap(int m, int n)
{ int temp;
  temp=m;
  m= n;
  n=temp;
}
int main()
{ int a=5, b=10;
  cout<<"a"<<a<<" b"<<b<<endl;
  swap(a, b);
  cout<<"a"<<a<<" b"<<b<<endl;
  return 0;
}

```

程序运行结果如下:

```

a=5 b=10
a=5 b=10

```

可见, 采用变量名作为函数参数, 调用函数 `swap` 后, 形参 `m` 和 `n` 的值被交换了。但是实参 `a` 和 `b` 的值没有交换, 仍是 5 和 10, 也就是说 `m` 和 `n` 值的改变不会影响 `a` 和 `b` 的值。

(2) 指针变量作为函数参数。这时实参传给形参的是实参变量的地址, 即“传址调用”。

这种传递是双向的，在执行函数期间形参值发生的变化传回给实参，因为在调用函数时，形参和实参占有同一个存储单元。下面的程序能够实现两个变量值的互换。

例 2.19 指针变量作为函数参数。

```
#include<iostream>
using namespace std;
void swap(int *m, int *n)
{ int temp;
  temp=*m;
  *m= *n;
  *n=temp;
}
int main()
{ int a=5, b=10;
  cout<<"a="<<a<<" b="<<b<<endl;
  swap(&a, &b);
  cout<<"a="<<a<<" b="<<b<<endl;
  return 0;
}
```

程序运行结果如下：

```
a=5 b=10
a=10 b=5
```

可见，采用指针变量作为函数参数，调用函数 swap 后，a 和 b 的值被交换了。

除了采用指针变量作为函数参数的方式外，C++还提供了引用作为函数参数。请看下面的例子。

(3) 引用作为函数参数。C++提供了向函数传递数据的第 3 种方法，把变量的引用作为函数形参，即传送变量的别名。这时实参传给形参的是实参变量的地址，即“传址调用”。这种传递也是双向的，形参值发生的变化传回给实参，因为在调用函数时，形参变量是实参变量的引用（别名），它们占有同一个存储单元。

例 2.20 引用作为函数的参数。

```
#include<iostream>
using namespace std;
void swap(int &m, int &n)           //形参 m 和 n 是整数类型变量的引用
{ int temp;
  temp=m;
  m=n;
  n=temp;
}
int main()
{ int a=5, b=10;
  cout<<"a="<<a<<" b="<<b<<endl;
  swap(a, b);                      //实参 a 和 b 是整型变量，可以通过引用来修改实参 a 和 b 的值
  cout<<"a="<<a<<" b="<<b<<endl;
  return 0;
}
```

程序运行结果如下：

```
a=5 b=10
```

```
a=10 b=5
```

当程序中调用函数 swap 时, 实参 a 和 b 分别初始化引用 m 和 n, 所以 m 和 n 分别是变量 a 和 b 的别名, 对 m 和 n 的访问就是对 a 和 b 的访问。调用函数 swap 后, 引用 m 和 n 的值被交换了, 所以变量 a 和 b 的值也随着交换了。

尽管通过引用作为函数参数产生的效果同采用指针变量作为函数参数的效果是一样的, 但引用作为函数参数更清楚简单。采用这种方法, 函数的形参前不需要间接引用运算符 “*”, 函数调用时实参是变量。C++ 主张采用引用作为函数参数, 因为这种方法容易且不易出错。

3. 对引用的进一步说明

下面再对使用引用的一些细节作进一步的讨论。

(1) 不能建立引用的数组。例如:

```
int a[4]="abcd";
int &ra[4]=a;    //错误, 不能建立引用的数组
```

企图建立一个包含 4 个元素的引用的数组, 这样是不行的, 数组名 a 只代表数组首元素的地址, 本身并不是一个占有存储空间的变量。

(2) 不能建立引用的引用, 不能建立指向引用的指针。引用本身不是一种数据类型, 所以没有引用的引用, 也没有引用的指针。例如:

```
int n=3;
int &&r=n;        //错误, 不能建立引用的引用
int *p=n;        //错误, 不能建立指向引用的指针
```

(3) 可以将引用的地址赋给一个指针, 此时指针指向的是原来的变量。例如:

```
int a=50;        //定义 a 是整型变量
int &b=a;         //声明 b 是整型变量 a 的引用
int *p=&b;        //指针变量 p 指向变量 a 的引用 b, 相当于指向 a
```

则 p 中保存的是变量 a 的地址, 其作用与下面一行相同, 即

```
int *p=&a;
```

如果输出 *p 的值, 就是 b 的值, 也就是 a 的值。

(4) 可以用 const 对引用加以限定, 不允许改变该引用的值。例如:

```
int a=5;          //定义整型变量 a, 初值为 5
const int &b=a     //声明常引用 b
b=3;              //错误, 不允许改变常引用 b 的值
```

但是它不阻止改变引用所代表的变量的值, 例如:

```
a=10;
```

此时输出 a 和 b 的值都是 10。

有时希望在函数中保护形参的值不被改变, 这时采用常引用作为函数形参时是很有用的。例如, 希望通过函数 i_Max 求出整型数组 a[200] 中的最大值, 函数原型应该是:

```
int i_Max(const int* ptr);
```

调用时的格式可以是:

```
i_Max(a);
```

这样做的目的是确保形参数组的数据不被破坏，即在函数中对数组元素的操作只许读，而不许写。

(5) 尽管引用运算符与地址操作符使用相同的符号“&”，但是它们是不一样的。引用仅在声明时带有引用运算符“&”，以后就像普通变量一样使用，不能再带“&”。其他场合使用的“&”都是地址操作符。例如：

```
int j=5;
int &i=j;           //声明引用 i, "&"为引用运算符
i=123;             //使用引用 i, 不带引用运算符
int *pi=&i;         //在此, "&"为地址操作符
cout<<&pi;          //在此, "&"为地址操作符
```

实 验

实验目的和要求

1. 熟悉 Visual C++ 6.0 的集成开发环境。
2. 学会使用 Visual C++ 6.0 编辑、编译、连接和运行 C++单文件程序的方法。
3. 初步了解 C++源程序的基本结构，学会使用简单的输入输出操作。
4. 了解 C++在非面向对象方面对 C 语言功能的扩充与增强。

实验内容和步骤

1. 编辑、编译、连接和运行以下的 C++单文件程序。

```
//test2_1.cpp
#include<iostream>
using namespace std;
int main()
{ cout<<"Hello!\n";
  cout<<"This is a program."<<endl;
  return 0;
}
```

2. 输入以下程序，进行编译，如果有错误，请修改程序，直到没有错误，然后进行连接和运行，并分析运行结果。

(1)

```
//test2_2_1_1.cpp
int main()
{ cout<<"Hello!\n";
  cout<<"Welcome to C++!"
}
```

(2)

```
//test2_2_2_1.cpp
#include<iostream>
using namespace std;
int main()
{ int x, y;
```

```

x=5;
y=6;
int z=x*y;
cout<<"x*y="<<z<<endl;
return 0;
}

```

(3)

```

//test2_2_3_1.cpp
#include<iostream>
using namespace std;
int main()
{ void fun(int, int&);
  int x, y;
  fun(3, x);
  fun(4, y);
  cout<<"x+y="<<x+y<<endl;
  return 0;
}
void fun(int m, int &n)
{ n=m*5 }

```

3. 编写一个程序，用来分别求 2 个整数、3 个整数、2 个双精度数和 3 个双精度数的最大值。要求使用重载函数来完成。

4. 编写一个程序，任意从键盘输入两个字符，能将它们按由大到小的顺序输出。要求程序中有一个交换两个字符的函数，其形参是变量的引用。

5. 编写一个程序，声明一个双精度型指针变量，使用运算符 `new` 动态分配一个 `double` 型存储区，将首地址赋给该指针变量，并输入一个数到该存储区中。计算以该数为半径的圆的面积，并在屏幕上显示出来，最后使用运算符 `delete` 释放该空间。

习 题

【2.1】简述 C++ 的主要特点。

【2.2】下面是一个 C 语言程序，请改写它，使之采用 C++ 风格的 I/O 语句。

```

#include<stdio.h>
double circle(double r)                //定义函数 circle
{ return 3.14*r*r;
}
double triangle(double h, double w)    //定义函数 triangle
{ return 0.5*h*w;
}
int main()                             //定义主函数 main
{ double r, h, w;
  double cs, ts;
  printf("Input r, h, w: ");
}

```

```

scanf("%lf%lf%lf", &r, &h, &w);           //输入圆的半径和三角形高和底的值
cs= circle(r);                             //调用函数 circle
ts= triangle(h, w);                        //调用函数 triangle
printf("The area of circle is:%f\n", cs);   //输出圆的面积
printf("The area of triangle is: %f\n", ts); //输出三角形的面积
return 0;
}

```

【2.3】通常只有规模很小而使用频繁的函数适宜定义为 ()。

- A. 重载函数 B. 内联函数 C. 递归函数 D. 嵌套函数

【2.4】使用关键字 (), 声明或定义的函数为一个内联函数。

- A. static B. const C. inline D. extern

【2.5】下列描述中, () 是错误的。

- A. 内联函数主要解决程序的运行效率问题
B. 内联函数的定义必须出现在内联函数第 1 次被调用之前
C. 内联函数中可以包括各种语句
D. 对内联函数不可以进行异常接口声明

【2.6】在 C++ 中, 下列设置参数默认值的描述中, 正确的是 ()。

- A. 设置参数默认值时, 应该全部参数都设置
B. 设置参数默认值, 只能在函数定义时进行
C. 设置参数默认值时, 应该先设置右边的, 再设置左边的
D. 程序中有函数重载, 就不能设置参数默认值

【2.7】有函数原型 `void test(int a, int b=7, char c='*')`, 下面的函数调用中, 属于不合法调用的是 ()。

- A. `test(5);` B. `test(5, 8);` C. `test(6, "#");` D. `test(0, 0, '*');`

【2.8】函数重载的条件是 ()。

- A. 函数名相同, 但形参的个数或类型不同
B. 函数名相同, 具有相同的参数个数, 但返回值的类型不同
C. 函数名不同, 但形参的个数或类型相同
D. 函数名相同, 并且函数的返回类型相同

【2.9】重载函数在调用时选择的依据中, () 是错误的。

- A. 函数名字 B. 函数的返回类型
C. 参数个数 D. 参数的类型

【2.10】下面的函数声明中, () 是 `void BC(int a, int b);` 的重载函数。

- A. `int BC(int a, int b);` B. `void BC(int x, int y);`
C. `float BC(int a, int b);` D. `void BC(float a, float b, float c);`

【2.11】关于 `new` 运算符的下列描述中, () 是错误的。

- A. 它可以用来动态创建对象和对象数组
B. 使用它创建的对象或对象数组可以使用运算符 `delete` 删除
C. 使用它创建对象时要调用构造函数
D. 使用它创建对象数组时必须指定初始值

【2.12】关于 delete 运算符的下列描述中, () 是错误的。

- A. 它必须用于 new 返回的指针
- B. 使用它删除对象时要调用析构函数
- C. 对一个指针可以使用多次该运算符
- D. 指针名前只有一对方括号符号, 不管所删除数组的维数

【2.13】下列语句中, () 是错误的。

- A. int *p=new int(5);
- B. int *p=new int[5];
- C. int *p=new int;
- D. int *p=new int[5](0);

【2.14】假设已经有定义 “const char *const name=“chen”;;”, 下面的语句中正确的是 ()。

- A. name[3]='a';
- B. name="lin";
- C. name=new char[5];
- D. cout<<name[3];

【2.15】假设已经有定义 “char * const name=“chen”;;”, 下面的语句中正确的是 ()。

- A. name[3]='q';
- B. name="lin";
- C. name=new char[5];
- D. name=new char('q');

【2.16】假设已经有定义 “const char *name=“chen”;;”, 下面的语句中错误的是 ()。

- A. name[3]='q';
- B. name="lin";
- C. name=new char[5];
- D. name=new char('q');

【2.17】下面对引用的描述中, () 是错误的。

- A. 引用是某个变量的别名
- B. 建立引用时, 要对它初始化
- C. 对引用初始化可以使用任意类型的变量
- D. 引用与其代表的变量具有相同的地址

【2.18】下面的类型声明中正确的是 ()。

- A. int &a[4];
- B. int &*p;
- C. int &&q;
- D. int i, *p=&i;

【2.19】已知 “int m=10;” 在下列表示引用的方法中, () 是正确的。

- A. int &x=m;
- B. int &y=10;
- C. int &z;
- D. float &t=&m;

【2.20】下列描述中关于引用调用的是 ()。

- A. 形参是指针, 实参是地址值
- B. 形参是引用, 实参是变量
- C. 形参和实参都是变量
- D. 形参和实参都是数组名

【2.21】回答问题。

(1) 以下两个函数原型是否等价:

```
float fun(int a, float b, char *c);
```

```
float fun(int, float, char *c);
```

(2) 以下两个函数是否等价:

```
float fun(int a, float b, char *c)
```

```
float fun(int, float, char *)
```

【2.22】以下这个简短的 C++ 程序不可能编译通过, 为什么?

```
#include<iostream>
using namespace std;
int main()
{ int a, b, c;
  cout<<"Enter two numbers: ";
  cin>>a>>b;
```

```
c=sum(a, b);  
cout<<"sum is:"<<c;  
return 0;  
}  
sum(int a, int b)  
{ return a+b;  
}
```

【2.23】分析下面程序运行的结果。

```
#include<iostream>  
using namespace std;  
int main()  
{ int x, y;  
  x=20;  
  y=45;  
  cout<<"x+y="<<x+y<<endl;  
  return 0;  
}
```

【2.24】写出下列程序的运行结果。

```
#include<iostream>  
using namespace std;  
int i=15;  
int main()  
{ int i;  
  i=100;  
  ::i=i+1;  
  cout<<::i<<endl;  
  return 0;  
}
```

【2.25】写出下列程序的运行结果。

```
#include<iostream>  
using namespace std;  
void f(int &m, int n)  
{ int temp;  
  temp=m;  
  m=n;  
  n=temp;  
}  
int main()  
{ int a=5, b=10;  
  f(a, b);  
  cout<<a<<" "<<b<<endl;  
  return 0;  
}
```

【2.26】分析下面程序的运行结果。

```
#include<iostream>  
using namespace std;  
int &f(int &i)  
{ i+=10;  
  return i;  
}
```

```

}
int main()
{ int k=0;
  int &m=f(k);
  cout<<k<<endl;
  m=20;
  cout<<k<<endl;
  return 0;
}

```

【2.27】分析下面程序运行的结果。请先阅读程序，写出程序运行时应输出的结果，然后上机运行程序，验证自己分析的结果是否正确。

```

#include<iostream>
using namespace std;
int main()
{ int a, b, c;

  int f(int a1, int b1, int c1);
  cin>>a>>b>>c;
  c=f(a, b, c);
  cout<<c<<endl;
  return 0;
}

int f(int a1, int b1, int c1)
{ int m;
  if(a1>b1) m=a1;
  else m=b1;
  if(c1>m) m=c1;
  return m;
}

```

【2.28】编写一个完整的C++程序，使用系统函数 $\text{pow}(x, y)$ 计算 x^y 的值，注意包含头文件 `cmath`。

【2.29】编写一个C++风格的程序，用动态分配空间的方法计算 Fibonacci 数列的前 20 项并存储到动态分配的空间中。

【2.30】编写一个C++风格的程序，建立一个 `sroot` 函数，返回其参数的二次方根。重载函数 `sroot` 3 次，让它返回整数、长整数与双精度数的二次方根（计算二次方根时，可以使用标准库函数 `sqrt`）。

第3章

类和对象

类（class）是一种用户自定义的复杂数据类型，它是将不同类型的数据和与这些数据相关的操作封装在一起的集合体。C++对 C 语言的改进，最重要的就是增加了“类”这种类型。所以 C++开始时被称为“带类的 C”。类是 C++中最重要、最基本的概念，它是面向对象程序设计的基础。本章主要介绍类的构成、成员函数、对象的定义和使用、构造函数与析构函数等内容。

3.1 类的构成

3.1.1 从结构体到类

结构体是 C 语言的一种自定义的数据类型，它把相关联的数据元素组成一个单独的统一体。例如，下面声明了一个日期结构体：

```
struct Date{
    int year;
    int month;
    int day;
};
```

结构体 Date 中包含了 3 个数据元素：year、month 和 day，分别表示年、月和日。以下是这个例子的完整程序。

例 3.1 有关日期结构体的例子。

```
#include<iostream>
using namespace std;
struct Date{           //声明了一个名为 Date 的结构体
    int year;
    int month;
    int day;
};
int main()
{ Date datel;
  datel.year=2010;    //可以在结构体外直接访问数据 year
  datel.month=8;      //可以在结构体外直接访问数据 month
  datel.day=25;       //可以在结构体外直接访问数据 day
  cout<<datel.year<<". "<<datel.month<<". "<<datel.day<<endl;
  return 0;
}
```

程序的运行结果如下：

2010.8.25

C 语言中的结构体存在一些缺点。例如，一旦建立了一个结构体变量，就可以在结构体外直接访问数据。如上例 main 函数中，我们可以用赋值语句随意访问结构体变量中的数据 year、month 和 day。但是在现实世界中有些数据是不允许随意访问的。换句话说，不同的使用者对数据访问的权限是不一样的。例如，一个人的出生日期是不能随意访问修改的。可见，在 C 语言结构体中的数据是很不安全的，C 语言结构体无法对数据进行保护和权限控制。C 语言结构体中的数据与对这些数据进行的操作是分离的，没有把这些相关的数据和操作（通常用函数实现）构成一个整体进行封装，因此使程序的复杂性很难控制，维护数据和处理数据要花费很大的精力，使传统程序难以重用，严重影响了软件的生产效率。

在 C++ 中，引入了类的概念，它能克服 C 语言结构体的这些缺点。C++ 中的类将数据和与之相关的函数封装在一起形成一个整体，具有良好的外部接口，可以防止数据未经授权的访问，提供了模块间的独立性。

3.1.2 类的构成

C++ 提供了一种比结构体类型更安全有效的数据类型——类。类是 C++ 的一个最重要的特性。类与结构体的扩充形式十分相似，类声明中的内容包括数据和函数，分别称为数据成员和成员函数。按访问权限划分，数据成员和成员函数又可分为公有和私有两种，分别是公有数据成员与公有成员函数、私有数据成员与私有成员函数（在 C++ 中，还有保护数据成员和保护成员函数。关于保护成员将在第 5 章详细介绍。）

类声明的一般格式如下：

```
class 类名{
public:
    公有数据成员;
    公有成员函数;
private:
    私有数据成员;
    私有成员函数;
};
```

类的声明由关键字 class 打头，后跟类名，花括号中是类体，最后以一个分号“;”结束。这里，还是以日期为例，用一个类 Date 来描述日期，其形式如下：

```
class Date {
public:
    void setDate(int y, int m, int d);    //公有成员函数
    void showDate();                    //公有成员函数
private:
    int year;                            //私有数据成员
    int month;                            //私有数据成员
    int day;                             //私有数据成员
};
```

在一般情况下,类体中仅给出成员函数原型,而把函数体的定义放在类体外实现。成员函数的具体定义将在后续章节讨论。

在类 `Date` 中,数据成员和成员函数分别属于 `private` 部分和 `public` 部分。为什么要把它们分为不同的部分呢?这是因为它们的性质不同,或者说它们有不同的访问权限。

`private` 部分称为类的私有部分,这一部分的数据成员和成员函数称为类的私有成员。私有成员只能由本类的成员函数访问,而类外部的任何访问都是非法的。这样,私有成员就整个隐藏在类中,在类的外部根本就无法访问,实现了访问权限的有效控制。在类 `Date` 中就声明了 3 个只能由内部成员函数访问的数据成员,即 `year`、`month` 和 `day`。

`public` 部分称为类的公有部分,这部分的数据成员和成员函数称为类的公有成员。公有成员可以由程序中的函数(包括类内和类外)访问,即它对外是完全开放的。公有成员函数是类与外界接口,来自类外部的访问需要通过这种接口来进行。例如,在类 `Date` 中声明了设置日期成员函数 `setDate` 和日期显示成员函数 `showDate`,它们都是公有成员函数,类外部若想对类 `Date` 的数据进行操作,只能通过这两个函数来实现。

说明:

(1)除了 `private` 和 `public` 之外,类中的成员还可以用另一个关键字 `protected` 来说明。被 `protected` 说明的数据成员和成员函数称为保护成员。保护成员可以由本类的成员函数访问,也可以由本类的派生类的成员函数访问,而类外的任何访问都是非法的,即它是半隐藏的,关于保护成员将在第 4 章详细介绍。

(2)对一个具体的类来讲,类声明格式中的 3 个部分并非一定要全有,但至少要有其中的一个部分。一般情况下,一个类的数据成员应该声明为私有成员,成员函数声明为公有成员。这样,内部的数据隐藏在类中,在类前面的外部无法直接访问,使数据得到有效的保护,也不会对该类以外的其余部分造成影响,程序模块之间的相互作用就被降低到最小。

(3)类声明中的 `private`、`protected` 和 `public` 3 个关键字可以按任意顺序出现任意次。但是,如果把所有的私有成员、保护成员和公有成员归类放在一起,程序将更加清晰。

(4)C++规定,在默认情况下(即没有指定成员是私有、保护或公有时),类中的成员是私有的。C++结构体中的成员同样可以分为私有成员和公有成员,但是在默认情况下,结构体中的成员是公有的。因此,在例 3.1 结构体 `Date` 中的成员默认为公有的,即数据成员 `year`、`month` 和 `day` 是公有的,结构体外的变量 `date1` 能够对它们直接进行访问,所以以下 4 条语句是可以正常运行的:

```
date1.year=2010;
date1.month=8;
date1.day=25;
cout<<date1.year<<". "<<date1.month<<". "<<date1.day<<endl;
```

但是,如果将例 3.1 中的 `struct` 改为 `class`,将该例中的结构体改成类,这时类 `Date` 中的成员都默认为私有的,即数据成员 `year`、`month` 和 `day` 是私有的,类外的对象 `date1` 不能对它们直接进行访问。

例 3.2 将例 3.1 中的结构体 `struct` 改为类 `class`。

```
#include<iostream>
using namespace std;
class Date{
    int year;
    //将例 3.1 中的结构体 struct 改为类 class
    //私有数据成员
```

```

    int month;                //私有数据成员
    int day;                  //私有数据成员
};
int main()
{
    Date dater;
    dater.year=2010;          //错误，在类外不能访问私有数据成员 year
    dater.month=8;            //错误，在类外不能访问私有数据成员 month
    dater.day=25;             //错误，在类外不能访问私有数据成员 day
    cout<<dater.year<<". "<<dater.month<<". "<<dater.day<<endl;
                                //错误，不能访问私有数据成员

    return 0;
}

```

那么如何才能访问类中的私有数据成员呢？本书将在 3.2 节作详细的介绍。

(5) 有些程序员主张将所有的私有成员放在其他成员的前面，因为一旦用户忘记了使用说明符 `private`，由于默认值是 `private`，这将使用户的数据仍然得到保护。另一些程序员主张将公有成员放在最前面，这样可以使用户将注意力集中在能被外界调用的成员函数上，使用户思路更清晰一些。不论 `private` 部分放在前面，还是 `public` 部分放在前面，类的作用是完全相同的。

(6) 不能在类声明中给数据成员赋初值。例如：

```

class Date {
public:
    int year=2009;            //错误
    int month=6;              //错误
    int day=16;               //错误
};

```

C++规定，只有在类的对象定义之后才能给数据成员赋初值。

3.2 成员函数的定义

类的成员函数是函数的一种，它也有函数名、返回值类型和参数表，它的用法与普通函数基本上是一样的，只是它属于一个类的成员。成员函数可以访问本类中任何成员（包括公有的、私有的和保护了的）。成员函数可以被指定为私有的(`private`)、公有的(`public`)和保护了的(`protected`)。其中，私有的成员函数只能被本类中其他成员函数调用，不能被类外的对象调用；公有的成员函数既可以被本类的成员函数访问，也可以在类外被该类的对象访问。保护的成员函数的内容将在第 5 章详细介绍。

在 C++ 程序设计中，成员函数既可以定义成普通的成员函数（即非内联的成员函数），也可以定义成内联成员函数。

3.2.1 普通成员函数的定义

定义成员函数的第 1 种方式是，将成员函数以普通成员函数（即非内联的成员函数）的形式进行定义。在类声明中只给出成员函数的原型，而成员函数的定义写在类的外部。这种

成员函数在类外定义的一般形式是:

返回值类型 类名::成员函数名(参数表)

```
{
    函数体
}
```

例如,表示日期的类 Date 可声明如下:

```
class Date{
public:
    void setDate(int y, int m, int d);           //设置日期的成员函数 setDate 的函数原型
    void showDate();                             //显示日期的成员函数 showDate 的函数原型
private:
    int year;
    int month;
    int day;
};
void Date::setDate(int y, int m, int d)         //在类外定义成员函数 setDate
{
    year=y;
    month=m;
    day=d;
}
void Date::showDate()                          //在类外定义成员函数 setDate
{
    cout<<year<<". "<<month<<". "<<day<<endl;
}
```

从这个例子可以看出,虽然函数 setDate 和 showDate 在类外部定义,但它们属于类 Date 的成员函数,它们可以直接访问类 Date 中的私有数据成员 year、month 和 day。

说明:

(1) 在类外定义成员函数时,必须在成员函数名之前缀上类名,在类名和函数名之间应加上作用域运算符“::”,用于声明这个成员函数是属于哪个类的。例如,上面例子中的“Date::”,说明这些成员函数是属于类 Date 的。如果在函数名前没有类名,或既无类名又无作用域运算符“::”,如

```
::setDate()或 setDate()
```

则表示 setDate 函数不属于任何类,这个函数不是成员函数,而是普通的函数。

(2) 在类声明中,成员函数原型的参数表中可以不说明参数的名字,而只说明它们的类型。例如:

```
void setDate(int, int, int);
```

但是,在类外定义成员函数时,不但要说明参数表中参数的类型,还必须要指出其参数名。

(3) 采用“在类声明中只给出成员函数的原型,而将成员函数的定义放在类的外部”的定义方式,是 C++程序设计的良好习惯。这种方式不仅可以减少类体的长度,使类的声明简洁明了,便于阅读,而且有助于把类的接口和类的实现细节相分离,隐藏了执行的细节。

3.2.2 内联成员函数的定义

定义成员函数的第2种方式是，将成员函数以内联函数的形式进行定义。在C++中，可以用下面两种格式定义内联成员函数：

(1) 隐式声明。将成员函数直接定义在类的内部。例如：

```
class Date{
public:
    void setDate(int y, int m, int d)           //成员函数 setDate 直接定义在类的内部
    { year=y;
      month=m;
      day=d;
    }
    void Date::showDate()                      //成员函数 showDate 直接定义在类的内部
    { cout<<year<<". "<<month<<". "<<day<<endl;
    }
private:
    int year;
    int month;
    int day;
};
```

此时，函数 `setDate` 和 `showDate` 就是隐含的内联成员函数。内联函数的调用类似宏指令的扩展，它直接在调用处扩展其代码，而不进行一般函数的调用操作。这种定义内联成员函数的方法没有使用关键字 `inline` 进行声明，因此这种定义内联成员函数的方法称为隐式定义。

(2) 显式声明。在类声明中只给出成员函数的原型，而将成员函数的定义放在类的外部。但是在类内函数原型声明前或在类外定义成员函数前冠以关键字“`inline`”，以此显式地说明这是一个内联函数。这种定义内联成员函数的方法称为显式定义。

例如，上面的例子改为显式声明可变成如下形式：

```
class Date{
public:
    inline void setDate(int y, int m, int d);    //声明成员函数 setDate 为内联函数
    inline void showDate();                    //声明成员函数 showDate 为内联函数
private:
    int year;
    int month;
    int day;
};
inline void Date::setDate(int y, int m, int d) //在类外定义此函数为内联函数
{ year=y;
  month=m;
  day=d;
}
inline void Date::showDate()                  // 在类外定义此函数为内联函数
{ cout<<year<<". "<<month<<". "<<day<<endl;
}
```

也可以定义如下形式：

```
class Date{
public:
```

```

        void setDate(int y, int m, int d);           //在此, 不加关键字 inline
        void showDate();                           //在此, 不加关键字 inline
    private:
        int year;
        int month;
        int day;
};

inline void Date::setDate(int y, int m, int d)      //在此, 加上关键字 inline
{
    year=y;
    month=m;
    day=d;
}

inline void Date::showDate()                       //在此, 加上关键字 inline
{
    cout<<year<<". "<<month<<". "<<day<<endl;
}

```

说明:

在类中, 使用 inline 定义内联函数时, 必须将类的声明和内联成员函数的定义都放在同一个文件 (或同一个头文件) 中, 否则编译时无法进行代码置换。

3.3 对象的定义和使用

3.3.1 类与对象的关系

通常我们把具有共同属性和行为的事物所构成的集合称为类。在 C++ 中, 可以把具有相同数据和相同操作集的对象看成属于同一类。

一个类也就是用户声明的一个数据类型。每一种数据类型 (包括基本数据类型和自定义类型) 都是对一类数据的抽象, 在程序中定义的每一个变量都是其所属数据类型的一个实例。类的对象可以看成是该类类型的一个实例, 定义一个对象和定义一个一般变量相似。

在 C++ 中, 类与对象间的关系, 可以用数据类型 int 和整型变量 i 之间的关系来类比。类类型和 int 类型均代表的是一般的概念, 而对象和整型变量却是代表具体的东西。正像定义 int 类型的变量一样, 也可以定义类的变量。C++ 把类的变量称为类的对象, 对象也称为类的实例。

3.3.2 对象的定义

在 C++ 中, 有以下两种方法定义对象。

(1) 在声明类的同时, 直接定义对象, 即在声明类的右花括号 “}” 后, 直接写出属于该类的对象名表。例如:

```

class Date{
public:
    void setDate(int y, int m, int d)              //成员函数 setDate
    {
        year=y;
        month=m;
        day=d;
    }
}

```

```

void Date::showDate()                //成员函数 showDate
{ cout<<year<<". "<<month<<". "<<day<<endl;
}
private:
    int year;
    int month;
    int day;
} date1, date2;

```

在声明类 Date 的同时，直接定义了对象 date1 和 date2。

(2) 声明了类之后，在使用时再定义对象。定义对象的格式与定义基本数据类型变量的格式类似，其一般形式如下：

类名 对象名 1, 对象名 2, …;

例如：

```
Date date1, date2;
```

此时定义了 date1 和 date2 为 Date 类的两个对象。

说明：

声明了一个类便声明了一种类型，它并不接收和存储具体的值，只作为生成具体对象的一种“样板”，只有定义了对象后，系统才为对象分配存储空间，以存放对象中的成员。

3.3.3 对象中成员的访问

不论是数据成员，还是成员函数，只要是公有的，在类的外部可以通过类的对象进行访问。访问对象中的成员通常有以下 3 种方法。

1. 通过对象名和对象选择符访问对象中的成员

其一般形式是：

对象名.数据成员名

或

对象名.成员函数名[[实参表]]

其中“.”称为对象选择符，简称点运算符。

例 3.3 通过对象名和对象选择符访问对象中的成员。

```

#include<iostream>
using namespace std;
class Date{
public:
    void setDate(int y, int m, int d);
    void showDate();
private:
    int year;
    int month;
    int day;
};
void Date::setDate(int y, int m, int d)
{ year=y;
  month=m;
  day=d;
}

```

```

}
void Date::showDate()
{ cout<<year<<". "<<month<<". "<<day<<endl; }
int main()
{ Date date1, date2;
  cout<<"Date1 set and output:"<<endl;
  date1.setDate(2010, 4, 28);           //调用对象 date1 的公有成员函数 setDate, 给
                                         //date1 的私有数据成员赋值
  date1.showDate();                    //调用对象 date1 公有成员函数 showDate, 显示
                                         //date1 的私有数据成员

  cout<<"Date2 set and output:"<<endl;
  date2.setDate(2010, 11, 14);         //调用对象 date2 的公有成员函数 setDate, 给
                                         //date2 的私有数据成员赋值
  date2.showDate();                    //调用对象 date2 公有成员函数 showDate, 显示
                                         //date2 的私有数据成员

  return 0;
}

```

程序运行结果如下:

Date1 set and output:

2010.4.28

Date2 set and output:

2010.11.14

说明:

一定要注意所访问的成员是公有的 (public) 还是私有的 (private)。在类外只能访问公有 (public) 成员, 而不能访问私有 (private) 成员。如果将例 3.3 中的主程序改成下面的形式, 将出现编译错误:

```

int main()
{ Date date1;
  date1.year=2010;           //错误, 在类的外部不能访问私有数据成员 year
  date1.month=8;             //错误, 在类的外部不能访问私有数据成员 month
  date1.day=25;              //错误, 在类的外部不能访问私有数据成员 day
  cout<<date1.year<<". "<<date1.month<<". "<<date1.day<<endl;
                               //错误, 在类的外部不能访问私有数据成员

  return 0;
}

```

2. 通过指向对象的指针访问对象中的成员

在定义对象时, 若我们定义的是指向此对象的指针, 则访问此对象的成员时, 不能用 “.” 操作符, 而应使用 “->” 操作符, 例如:

```

class Date{
public:
  int year;                //公有数据成员
  ...
};
...
Date d, *ptr;              //定义对象 d 和指向类 Date 的指针变量 ptr

```

```
ptr=&d;                //使 ptr 指向对象 d
cout<<ptr->year;       //输出 ptr 指向对象中的成员 year
```

在此, `ptr->year` 表示 `ptr` 当前指向对象 `d` 中的成员 `year`, 因为 `(*ptr)` 就是对象 `d`, `(*ptr).year` 表示的也就是对象 `d` 中的成员 `year`。所以, 在此

```
d.year
(*ptr).year
ptr->year }
```

三者是等价的

3. 通过对象的引用访问对象中的成员

如果为一个对象定义了一个引用, 也就是为这个对象起了一个别名。因此可以通过引用来访问对象中的成员, 其方法与通过对象名来访问对象中的成员是相同的。例如:

```
class Date{
public:
    int year;           //公有数据成员
};

Date d1;               //定义类 Date 的对象 d1
Date &d2=d1;           //定义类 Date 的引用 d2, 并用对象 d1 进行初始化
cout<<d1.year;         //输出对象 d1 中的数据成员 year
cout<<d2.year;         //输出对象 d2 中的数据成员 year
```

由于 `d2` 是 `d1` 的引用 (即 `d2` 和 `d1` 占有相同的存储单元), 因此 `d2.year` 和 `d1.year` 是相同的。

3.3.4 类的作用域和类成员的访问属性

类的作用域就是指在类的声明中的一对花括号所形成的作用域。一个类的所有成员都在该类的作用域内。在类的作用域内, 一个类的任何成员函数可以不受限制地访问该类中的其他成员。而在类作用域之外, 对该类的数据成员和成员函数的访问则要受到一定的限制, 有时甚至是不允许的, 这主要与类成员的访问属性有关。

下面, 我们归纳一下类成员的访问属性。类成员有 3 种访问属性, 即公有属性、私有属性和保护属性 (保护属性将在第 5 章介绍)。说明为公有的成员不但可以被类中成员函数访问, 还可在类的外部, 通过类的对象进行访问。说明为私有的成员只能被类中成员函数访问, 不能在类的外部, 通过类的对象进行访问。例如, 声明了以下一个类:

```
class Sample{
private:
    int i;
public:
    int j;
    void set(int i1, int j1)
    { i=i1;                //类的成员函数可以访问类的私有成员 i
      j=j1;                //类的成员函数可以访问类的公有成员 j
    }
};
```

在类的外部, 主函数 `main` 定义如下:

```
int main()
```

```

{ Sample a;           //定义类 Sample 的对象 a
  a.set(3, 5);        //在类外, 类 Sample 的对象 a 可以访问公有成员函数 set
  cout<<a.i<<endl;    //非法, 在类外, 类 Sample 的对象 a 不能访问类的私有成员 i
  cout<<a.j<<endl;    //合法, 在类外, 类 Sample 的对象 a 能够访问类的公有成员 j
  return 0;
}

```

通过上例可以说明, 在类的内部, 类 Sample 的成员函数可以访问类的私有成员 i 和公有成员 j。但是, 在类的外部, 类 Sample 的对象 a 可以访问类的公有成员 j, 而不能访问类的私有成员 i。

一般来说, 公有成员是类的对外接口, 而私有成员是类的内部数据和内部实现, 不希望外界访问。将类的成员划分为不同的访问级别有两个好处: 一是信息隐蔽, 即实现封装, 将类的内部数据与内部实现和对外接口分开, 这样使该类的外部程序不需要了解类的详细实现; 二是数据保护, 即将类的重要信息保护起来, 以免其他程序进行不恰当地修改。

3.4 构造函数与析构函数

构造函数和析构函数都是类的成员函数, 但它们都是特殊的成员函数, 执行特殊的功能, 而且这些函数的名字与类的名字有关。

3.4.1 对象的初始化和构造函数

我们知道, 在计算机中不同的数据类型分配的存储空间是不同的。类是一种用户自定义的类型, 它可能比较简单, 也可能很复杂。当声明一个类对象时, 编译程序需要为对象分配存储空间, 为数据成员赋初值, 即进行必要的初始化。如需要给例 3.3 的类 Date 的 year、month 和 day 赋初值, 但类声明体中不能给数据成员直接赋初值, 下面的写法是错误的:

```

class Date {
private:
    int year=2010;    //错误
    int month=6;      //错误
    int day=16;       //错误
};

```

那么怎么才能给类 Date 的 year、month 和 day 赋初值呢? 在例 3.3 的类 Date 中使用了一个 setDate 函数来实现, 在每次使用一个新的对象时调用一下该函数, 就可以对需要的数据成员进行初始化。例如:

```

class Date{
public:
    void setDate(int y, int m, int d);
    void showDate();
private:
    int year;
    int month;
    int day;
};

```

```

...
int main()
{ Date date1;
  date1.setDate(2010, 4, 28);      //调用对象 date1 的公有成员函数 setDate, 给
                                   //date1 的私有数据成员赋初值
  ...
  return 0;
}

```

但是这种方法既不方便也容易忘记,如果用户不小心忘记了调用函数 setDate 来初始化类对象,那么结果就可能出错。而 C++ 语言提供了一个更好的方法,即利用类的构造函数来初始化类的成员。

构造函数是一种特殊的成员函数,它主要用于为对象分配空间,进行初始化。构造函数的名字必须与类名相同,而不能由用户任意命名。它的特点:可以有任意类型的参数,有任何返回类型,不返回任何值。它不需要用户来调用,而是在建立对象时自动执行的。构造函数的功能是由用户定义的,用户根据对象初始化的要求设计函数体和函数参数。下面的例子中,我们为类 Date 定义一个构造函数。

例 3.4 为类 Date 定义一个构造函数。

```

class Date {
public:
    Date(int y, int m, int d);      //声明构造函数 Date 的原型
    void showDate();
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d)    //定义构造函数 Date
{ year=y;
  month=m;
  day=d;
}
void Date::showDate()
{ cout<<year<<". "<<month<<". "<<day<<endl;
}

```

上面声明的类名为 Date,其构造函数名也是 Date。构造函数的主要功能是给对象分配空间,进行初始化,即对数据成员赋初值,这些数据成员一般为私有成员。

在建立对象的同时,采用构造函数给数据成员赋初值,通常有以下两种形式。

形式 1: 类名 对象名([实参表]);

这里的“类名”与构造函数名相同,“实参表”是为构造函数提供的实际参数。

下面通过创建一个类 Date 的对象 date1,看一下构造函数是如何被调用的。

例 3.5 建立对象的同时,用构造函数给数据成员赋初值。

```

#include<iostream>
using namespace std;
class Date{
public:
    ~ Date(int y, int m, int d);    //声明构造函数的原型,构造函数的

```

//名字必须与类名相同

```

    void showDate();
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d)           //定义构造函数 Date
{ cout<<"Constructing..."<<endl;      //执行此语句时,表示构造函数被调用了
  year=y;
  month=m;
  day=d;
}
void Date::showDate()
{ cout<<year<<". "<<month<<". "<<day<<endl; }
int main()
{ Date date1(2010, 4, 28);               //定义类 Date 的对象 date1, 自动调用构造函数
                                          //给对象 date1 的数据成员赋初值
    date1.showDate();                    //调用成员函数 showDate, 显示 date1 的数据
    return 0;
}

```

程序运行结果如下:

Constructing...

2010.4.28

从上面的例子可看出,在 main 函数中,没有显式调用构造函数 Date 的语句。构造函数是在定义对象时被系统自动调用的。也就是说,在定义对象 date1 的同时,构造函数 Date 被自动调用执行,分别给数据成员 year、month 和 day 赋初值,并显示信息“constructing...”。这条信息的显示并不是必须的,在此,只是说明构造函数被调用了,帮助大家理解构造函数的使用方法。接着调用 date1 的成员函数 showDate,显示对象 date1 的数据。在执行 return 语句之后,主函数中的语句已执行完毕,对象 date1 的生命周期结束。

形式 2: 类名 *指针变量名=new 类名[[实参表]];

这是一种使用 new 运算符动态建立对象的方式。例如:

Date *pdate=new Date(2010, 4, 28);

这时编译系统会开辟一段可以存放一个 Date 类对象的内存空间,同时通过构造函数给数据成员赋初值。这个对象没有名字,称为无名对象。但是该对象有地址,这个地址存放在指针变量 pdate 中。访问用 new 运算符动态建立的对象一般是不用对象名的,而是通过指针访问的。例如:

pdate->showDate();

当用 new 建立的对象使用结束,不再需要它时,可以用 delete 运算符予以释放。例如:

delete pdate;

下面,将例 3.5 的主函数改成用这种方法来实现,其运行结果与原例题完全相同。

```

int main()
{ Date *pdate;

```



```

pdate=new Date(2010, 4, 28);
// 以上两条语句可合写成:Date *pdate=new Date(1998, 4, 28);

pdate->showDate();
delete pdate;
return 0;
}

```

说明:

(1) 构造函数没有返回值, 在定义构造函数时, 是不能说明它的类型的, 甚至说明为 void 类型也不行。例如, 构造函数不能写成:

```
void Date(int y, int m, int d);
```

(2) 与普通的成员函数一样, 构造函数的函数体可写在类体内, 也可写在类体外。如例 3.5 中的类 Date 的构造函数可以声明如下:

```

class Date{
public:
    Date(int y, int m, int d)    // 构造函数定义在类内
    { cout<<"Constructing..."<<endl;
      year=y;
      month=m;
      day=d;
    }
    ...
private:
    int year;
    int month;
    int day;
};

```

与普通的成员函数一样, 当构造函数直接定义在类内时, 系统将构造函数作为内联函数处理。

(3) 构造函数一般声明为公有成员, 但它不需要也不能像其他成员函数那样被显式地调用, 它是在定义对象的同时被自动调用的, 而且只执行一次。下面的用法是错误的:

```
date1.Date(2010, 4, 28);
```

(4) 构造函数是可以不带参数的。例如:

```

class A{
public:
    A()                //不带参数的构造函数
    { x=50;}
    ...
private:
    int x;
};

```

此时, 类 A 的构造函数没有带参数, 在 main 函数中可以采用如下方法定义对象:

```
A a;
```

在定义对象 a 的同时, 构造函数 A 被系统自动调用执行。执行结果是, 给私有数据成员 x 赋初值 50。

通常，不带参数的构造函数给对象赋的初始值是固定的。如果需要在建立一个对象时，通过传递某些参数，对其中的数据成员进行初始化，应该采用前面讲到的带参数的构造函数来解决。

(5) 在构造函数的函数体中不仅可以对数据成员赋初值，而且可以包含其他语句，例如，上面类 Date 构造函数中的语句：

```
cout<<"Constructing..."<<endl;
```

但是一般不提倡在构造函数中加入与初始化无关的内容，以保持程序的清晰。

3.4.2 用成员初始化表对数据成员初始化

在声明类时，对数据成员的初始化工作一般在构造函数中用赋值语句进行。例如，例 3.5 中定义的构造函数：

```
Date::Date(int y, int m, int d)           //在类外定义构造函数，用
{                                           //赋值语句对数据成员赋初值
    year=y;
    month=m;
    day=d;
}
```

C++还提供另一种初始化数据成员的方法——用成员初始化表来实现对数据成员的初始化。这种方法不在函数体内用赋值语句对数据成员初始化，而是在函数首都实现的。例如，以上面定义的构造函数为例，可以改写成：

```
Date::Date(int y, int m, int d):year(y), month(m), day(d)
{ }                                           // 使用成员初始化表对数据成员初始化
```

其中，函数首部末尾冒号后面的“year(y), month(m), day(d)”就是成员初始化表。上面的成员初始化表表示，用形参 y 的值初始化数据成员 year，用形参 m 的值初始化数据成员 month，用形参 d 的值初始化数据成员 day。

带有成员初始化表的构造函数的一般形式如下：

类名::构造函数名([参数表]):(成员初始化表)

```
{
    //构造函数体
}
```

成员初始化表的一般形式为：

数据成员名 1(初始值 1)，数据成员名 2(初始值 2)，...

成员初始化表写法方便、简练，尤其当需要初始化的数据成员较多时更显其优越性，很多程序人员喜欢用这种方法。使用成员初始化表后，例 3.5 可改写如下：

例 3.6 建立对象的同时，用带有成员初始化表的构造函数给数据成员赋初值。

```
#include<iostream>
using namespace std;
class Date{
public:
    Date(int y, int m, int d);           //声明构造函数的原型，构造
```

//函数的名字必须与类名相同

```

    void showDate();
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d):year(y), month(m), day(d) //定义构造函数 Date
{ } //使用成员初始化表对数据成员初始化
void Date::showDate()
{ cout<<year<<". "<<month<<". "<<day<<endl; }
int main()
{ Date date1(2010, 4, 28); //定义类 Date 的对象 date1, 自动调用构造函数
    //给对象 date1 的数据成员赋初值
    date1.showDate(); //调用成员函数 showDate, 显示 date1 的数据
    return 0;
}

```

程序运行结果如下:

2010.4.28

3.4.3 析构函数

析构函数也是一种特殊的成员函数。它执行与构造函数相反的操作,通常用于撤销对象时的一些清理任务,如释放分配给对象的内存空间等。析构函数有以下一些特点:

- (1) 析构函数与构造函数名字相同,但它前面必须加一个波浪号(~)。
- (2) 析构函数没有参数,也没有返回值,而且不能重载。因此在一个类中只能有一个析构函数。
- (3) 当撤销对象时,编译系统会自动地调用析构函数。

下面我们重新说明类 Date,使它既含有构造函数,又含有析构函数。

例 3.7 含有构造函数和析构函数的 Date 类。

```

#include<iostream>
using namespace std;
class Date{
public:
    Date(int y, int m, int d); // 声明构造函数
    ~Date(); // 声明析构函数
    void showDate();
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d) // 定义构造函数
{ cout<<"constructing..."<<endl;
    year=y;
    month=m;
    day=d;
}

```

```

}
Date::~Date()                // 定义析构函数
{ cout<<"destruting..."<<endl;
}
inline void Date::showDate()
{ cout<<year<<". "<<month<<". "<<day<<endl; }
int main()
{ Date date1(2010, 4, 28);    // 定义类 Date 的对象 date1, 调用
                              // date1 的构造函数, 初始化对象 date1
    date1.showDate();        // 调用成员函数 showDate, 显示 date1 的数据
    return 0;
}

```

在类 Date 中定义了构造函数和析构函数。在这两个函数中, 都含有一条输出语句, 显示相应函数被调用的信息, 目的是帮助初学者更好地理解构造函数和析构函数的使用方法。在执行主函数时先建立对象 date1, 在建立对象 date1 时调用构造函数, 对对象 date1 中的数据成员赋初值, 并显示信息“constructing...”, 然后调用 date1 的成员函数 showDate。在执行 return 语句之后, 主函数中的语句已执行完毕, 对象 date1 的生命周期结束, 在撤销对象 date1 时就要调用析构函数, 释放分配给对象 date1 的存储空间, 并显示信息“destruting...”。这条信息的显示并不是必须的, 在此, 只是说明析构函数被调用了, 帮助大家理解析构函数的使用方法。

程序运行结果如下:

```

constructing...
2010.4.28
destruting...

```

说明:

在以下情况, 当对象的生命周期结束时, 析构函数会被自动调用。

- (1) 如果定义了一个全局对象, 则在程序流程离开其作用域 (如 main 函数结束或调用 exit 函数) 时, 调用该全局对象的析构函数。
- (2) 如果一个对象被定义在一个函数体内, 则当这个函数被调用结束时, 该对象应该释放, 析构函数被自动调用。
- (3) 若一个对象是使用 new 运算符动态创建的, 在使用 delete 运算符释放它时, delete 会自动调用析构函数。

下面我们以学生类为例, 对类的声明、对象的定义和使用, 以及构造函数和析构函数的使用方法作一个较完整的介绍。

例 3.8 较完整的学生类示例。

```

#include<iostream>
#include<string>
using namespace std;
class Student{
public:
    Student(char *name1, char *stu_no1, float score1);    //声明构造函数
    ~Student();                                            //声明析构函数
    void disp();                                           //成员函数, 用于显示数据
}

```

```

private:
    char *name;           //学生姓名
    char *stu_no;         //学生学号
    float score;          //学生成绩
};

Student::Student(char *name1, char *stu_no1, float score1)    // 定义构造函数
{
    name=new char[strlen(name1)+1];
    strcpy(name, name1);
    stu_no=new char[strlen(stu_no1)+1];
    strcpy(stu_no, stu_no1);
    score=score1;
}

Student::~~Student()                                          //定义析构函数
{
    delete []name;
    delete []stu_no;
}

void Student::disp()
{
    cout<<"name: "<<name<<endl;
    cout<<"stu_no: "<<stu_no<<endl;
    cout<<"score: "<<score<<endl;
}

int main()
{
    Student stu1("Li ming", "20080201", 90);                //定义类 Student 的对象 stu1, 调用
                                                                //构造函数, 初始化对象 stu1
    stu1.disp();                                              //调用成员函数 disp, 显示 stu1 的数据
    Student stu2("Wang fun", "20080202", 85);                //定义类 Student 的对象 stu2, 调用
                                                                //构造函数, 初始化对象 stu1
    stu2.disp();                                              //调用成员函数 disp, 显示 stu2 的数据
    return 0;
}

```

本程序的运行结果如下:

```

name: Li ming
stu_no: 20080201
score: 90
name: Wang fun
stu_no: 20080202
score: 85

```

在本例中, 类 Student 中声明了 3 个私有数据成员: name、stu_no 和 score, 分别表示学生的姓名、学号和成绩; 声明了一个构造函数和一个析构函数, 以及数据显示成员函数 disp, 它们都是公有数据成员。在本程序中, 当执行语句

```
Student stu1("Li ming", "20080201", 90);
```

时, 定义了类 Student 的对象 stu1 和 stu2 时, 分别调用了构造函数。也就是说, 在定义类的对象时就自动调用构造函数进行对象的初始化。当程序结束, 对象撤销 stu1 和 stu2 时, 分别调用了析构函数, 释放由运算符 new 分配的内存空间。这是构造函数和析构函数常见的用法,

即在构造函数中用运算符 `new` 为字符串分配存储空间，最后在析构函数中用运算符 `delete` 释放已分配的存储空间。

3.4.4 默认的构造函数和默认的析构函数

1. 默认的构造函数

在实际应用中，通常需要给每个类定义构造函数。如果没有给类定义构造函数，则编译系统自动地生成一个默认构造函数。按照构造函数的规定，默认构造函数名与类名相同。默认构造函数的这种形式也可以显式地定义在类体中。

例如，例 3.3 的类 `Date` 中没有定义任何构造函数。在主程序中有如下的说明语句：

```
Date date1, date2;
```

这时，编译系统为类 `Date` 生成下述形式的构造函数：

```
Date::Date()
{ }
```

并使用这个默认的构造函数对 `date1` 和 `date2` 进行初始化。

这个默认的构造函数不带任何参数，它只能为对象开辟一个存储空间，而不能给对象中的数据成员赋初值，这时的初始值是随机数，程序运行时可能会造成错误。

例 3.9 一个没有对数据成员赋初值的例子。

```
#include<iostream>
using namespace std;
class Myclass{
public:
    int no;
};
int main()
{ Myclass a;
  cout<<a.no<<endl;
  return 0;
}
```

程序运行结果如下：

```
-858993460
```

不难看出，本例的运行结果是一个随机数，在实际使用时可能会造成错误。

说明：

(1) 对没有定义构造函数的类，其公有数据成员可以用初始值表进行初始化。请看以下例子。

例 3.10 用初始值表初始化公有数据成员。

```
#include<iostream>
using namespace std;
class Myclass{
public:
    char name[10];
    int no;
};
int main()
{ Myclass a={"chen", 25};
```

```

    cout<<a.name<<" "<<a.no<<endl;
    return 0;
}

```

在本例中，main()中创建了一个类 myclass 的对象 a，并将初始值表中的“chen”和 25 分别赋给 a.name 和 a.no。

程序运行结果如下：

```
chen 25
```

这种方法对结构体和数组的初始化较适合。

(2) 只要一个类定义了一个构造函数（不一定是无参构造函数），系统将不再给它提供默认的构造函数。

例 3.11 分析程序的运行结果。

```

#include<iostream>
using namespace std;
class Date{
public:
    Date(int y, int m, int d);      // 带有参数的构造函数
    void setDate(int y, int m, int d);
    void showDate();
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d)
{ year=y;
  month=m;
  day=d;
}
void Date::setDate(int y, int m, int d)
{ year=y;
  month=m;
  day=d;
}
inline void Date::showDate()
{ cout<<year<<"."<<month<<"."<<day<<endl; }
int main()
{ Date date1;
  date1.setDate(2010, 11, 14);
  date1.showDate();
  return 0;
}

```

也许，有些读者认为该程序的运行结果是：

```
2010.11.14
```

实际上，运行结果表明，这是一个错误的程序。错误的原因是，当类中定义了带有参数的构造函数后，系统将不再给它提供默认的构造函数。因此当我们定义类 Date 的对象 date1 时，找不到与之匹配的构造函数。

可以采用以下方法解决这个问题。

(1) 在类中增加如下一个无参数的构造函数:

```
Date()
{ }
```

(2) 将主函数改写成以下形式:

```
int main()
{ Date date1(2010, 11, 14);
  date1.showDate();
  return 0;
}
```

2. 默认的析构函数

每个类必须有一个析构函数。若没有显式地为一个类定义析构函数,编译系统会自动地生成一个默认的析构函数。

例如,编译系统为类 Date 生成默认的构造函数如下所示:

```
Date::~Date()
{ }
```

对于大多数类而言,默认的析构函数就能满足要求。但是,如果在一个对象完成其操作之前需要做一些内部处理,则应该显式地定义析构函数,以完成所需的操作。例如,例 3.8 中的析构函数:

```
Student::~Student()
{ delete []name;           //释放由运算符 new 分配的内存空间
  delete []stu_no;         //释放由运算符 new 分配的内存空间
}
```

3.4.5 带默认参数的构造函数

对于带参数的构造函数,在定义对象时必须给构造函数传递参数,否则构造函数将不被执行。但在实际使用中,有些构造函数的参数值大部分情况是相同的,只有在特殊情况下才需要改变它的参数值,例如,大学本科的学制一般默认为 4 年,女子学院学生的性别一般默认为“女”等。这时可以将其定义成带默认参数的构造函数。示例如下。

例 3.12 带有默认参数的构造函数。

```
#include<iostream>
using namespace std;
class Rectangle{
public:
  Rectangle(int len=10, int wid=10)           //在定义构造函数时指定默认参数值
  { length=len; width=wid; }
  int area()
  { return (length*width); }
private:
  int length, width;
};
int main()
{ Rectangle rec1;                             //没有给定实参
  cout<<"The area of rectangle1 is "<<rec1.area()<<endl;
  Rectangle rec2(15);                         //给定了 1 个实参
  cout<<"The area of rectangle2 is "<<rec2.area()<<endl;
```



```

Rectangle rec3(15, 20);           //给定了2个实参
cout<<"The area of rectangle3 is "<<rec3.area()<<endl;
return 0;
}

```

在类 `Rectangle` 中，构造函数的两个参数均含有默认参数值 10。因此，在定义对象时可根据需要使用其默认值。

由于定义对象 `rec1` 时，没有传递实参，所以 `length` 和 `width` 均取构造函数的默认值为其赋值，因此 `length` 和 `width` 的值均为 10，即

```
rec1.length=10; rec1.width=10;
```

在定义对象 `rec2` 时，只传递了一个参数，这个参数传递给了构造函数的第 1 个形参 `len`，而第 2 个形参 `wid` 未得到实参传过来的值，就取默认值 10，即

```
rec2.length=15; rec2.width=10;
```

在定义对象 `rec3` 时，传递了两个实参，这两个实参分别传给了 `length` 和 `width`，因此 `length` 取值为 15，`width` 取值为 20，即

```
rec3.length=15; rec3.width=20;
```

程序运行结果如下：

```

The area of rectangle1 is 100
The area of rectangle2 is 150
The area of rectangle3 is 300

```

3.4.6 构造函数的重载

在一个类中可以定义多个构造函数，以便对类对象提供不同的初始化方法，以适应不同的情况场合。这些构造函数具有相同的名字，而参数的个数或参数的类型有所不同。这称为构造函数的重载。

下面通过一个例子来了解怎样使用构造函数的重载。

例 3.13 构造函数重载的示例。

```

#include<iostream>
using namespace std;
class Rectangle{
public:
    Rectangle();           //声明无参数的构造函数
    Rectangle(int len, int wid); //声明带有参数的构造函数
    int area()
    { return (length*width); }
private:
    int length, width;
};

Rectangle::Rectangle() //定义无参数的构造函数
{ length=10; width=10;
}

Rectangle::Rectangle(int len, int wid) //定义带有参数的构造函数
{ length=len; width=wid; }

int main()

```

```

{ Rectangle rec1;                                //定义类 Date 的对象 rec1, 调用无参数的构造函数
cout<<"The area of rectangle1 is "<<rec1.area()<<endl;
Rectangle rec2(15, 20);                          //定义类 Date 的对象 rec2, 调用
                                                //带有参数的构造函数
cout<<"The area of rectangle2 is "<<rec2.area()<<endl;
return 0;
}

```

在类中定义了两个构造函数, 第 1 个构造函数是没有参数的, 第 2 个构造函数是带有参数的。

在本例的 main 函数中定义了两个对象。对象 rec1 没有传递参数, 所以定义对象 rec1 时, 调用无参数的构造函数; 定义对象 rec2 时, 传递了两个整型参数, 所以定义对象 rec2 时, 调用含两个整型参数的构造函数。

程序运行结果如下:

```

The area of rectangle1 is 100
The area of rectangle2 is 300

```

说明:

(1) 使用无参构造函数创建对象时, 应该用语句 “Rectangle rec1;”, 而不能用语句 “Rectangle rec1();”。因为语句 “Rectangle rec1();” 表示声明一个名为 rec1 的普通函数, 此函数的返回值为 Rectangle 类型。

(2) 在一个类中, 当无参数的构造函数和带默认参数的构造函数重载时, 有可能产生二义性。例如:

```

class X {
public:
    X();                                //无参数的构造函数
    X(int i=0);                        //带默认参数的构造函数
};

int main()
{ X one(10);                          //正确, 调用带默认参数的构造函数
  X two;                              //存在二义性
}

```

该例定义了两个构造函数 X, 其中一个没有参数, 另一个带有默认参数。创建对象 two 时, 由于没有给出参数, 它既可以调用无参数的构造函数, 也可以调用带默认参数的构造函数。这时, 编译系统无法确定应该调用哪一个构造函数, 因此产生了二义性。在实际应用时, 一定要注意避免这种情况。

3.5 对象的赋值与复制

3.5.1 对象赋值语句

如果有两个整型变量 x 和 y, 那么用语句 y=x, 就可以把变量 x 的值赋给变量 y。同类型

的对象之间也可以进行赋值，一个对象的值可以赋给另一个对象。这里所指对象的值是指对象中所有数据成员的值。例如，A 和 B 是同一类的两个对象，假设变量 A 已经存在，那么下述对象赋值语句把对象 A 的数据成员的值逐位拷贝给对象 B：

```
B=A
```

对象之间的赋值也是通过赋值运算符“=”进行的。本来，赋值运算符“=”只能用来对基本数据类型（如 int、float 等）的数据赋值，C++ 可以将其扩展为用于两个同类对象之间的赋值，这是通过对赋值运算符的重载实现的（关于运算符的重载将在第 7 章中介绍）。对象赋值的一般形式如下：

对象名 1=对象名 2;

下面我们看一个使用对象赋值语句的例子。

例 3.14 使用对象赋值语句的示例。

```
#include<iostream>
using namespace std;
class Rectangle{
public:
    Rectangle(int len=10, int wid=10)    //在定义构造函数时指定默认参数值
    { length=len; width=wid; }
    int area()
    { return (length*width); }
private:
    int length, width;
};
int main()
{ Rectangle rec1(15, 20), rec2;          //定义两个对象 rec1 和 rec2
  cout<<"The area of rectangle1 is "<<rec1.area()<<endl;
  rec2=rec1;                             //将对象 rec1 的值赋给对象 rec2
  cout<<"The area of rectangle2 is "<<rec2.area()<<endl;
  return 0;
}
```

在该程序中，语句：

```
rec2=rec1;
```

等价于语句：

```
rec2.length=rec1.length;
rec2.width=rec1.width;
```

因此，运行此程序将显示：

```
The area of rectangle1 is 300
The area of rectangle2 is 300
```

说明：

(1) 在使用对象赋值语句进行对象赋值时，两个对象的类型必须相同，如果对象的类型不同，编译时将出错。

(2) 两个对象之间的赋值，仅仅是对其中的数据成员赋值，而不对成员函数赋值。数据成员是占存储空间的，不同对象的数据成员占有不同的存储空间，而不同对象的成员函数是占有同一个函数代码段，无法对它们赋值。

(3) 当类中存在指针时,使用默认的赋值运算符函数进行对象赋值,可能会产生错误。使用时一定要注意,在此不作详细分析。

3.5.2 拷贝构造函数

拷贝构造函数是一种特殊的构造函数,其形参是本类对象的引用。拷贝构造函数的作用是,在建立一个新对象时,使用一个已经存在的对象去初始化这个新对象。例如:

```
Point p2(p1);
```

其作用是,在建立新对象 p2 时,用已经存在的对象 P1 去初始化新对象 p2,在这个过程中就要调用拷贝构造函数。

拷贝构造函数具有以下特点。

(1) 因为拷贝构造函数也是一种构造函数,所以其函数名与类名相同,并且该函数也没有返回值。

(2) 拷贝构造函数只有一个参数,并且是同类对象的引用。

(3) 每个类都必须有一个拷贝构造函数。程序员可以自定义拷贝构造函数,用于按照需要初始化新对象。如果程序员没有定义类的拷贝构造函数,系统就会自动生成产生一个默认拷贝构造函数,用于复制出数据成员值完全相同的新对象。

1. 自定义拷贝构造函数

自定义拷贝构造函数的一般形式如下:

类名::类名(const 类名 &对象名)

```
{
    拷贝构造函数的函数体
}
```

下面是一个用户自定义的拷贝构造函数:

```
class Rectangle{
public:
    Rectangle(int len, int wid)           // 构造函数
    { length=len;
      width=wid;
      cout<<"Using normal constructor\n";
    }
    Rectangle(const Rectangle& p)         // 自定义的拷贝构造函数
    { length=2*p.length;
      width=2*p.width;
      cout<<"Using copy constructor\n";
    }
    ...
private:
    int length, width;
};
```

假如, p1、p2 为类 Rectangle 的两个对象,且 p1 已经存在,则下述语句可以调用拷贝构造函数初始化 p2:

```
Rectangle p2(p1);
```

下面是使用这个自定义拷贝构造函数的完整程序。

例 3.15 自定义拷贝构造函数的使用。

```
#include<iostream>
using namespace std;
class Rectangle{
public:
    Rectangle(int len, int wid)           // 构造函数
    { length=len;
      width=wid;
      cout<<"Using normal constructor\n";
    }
    Rectangle(const Rectangle& p)         // 自定义的拷贝构造函数
    { length=2*p.length;
      width=2*p.width;
      cout<<"Using copy constructor\n";
    }
    void disp()
    { cout<<"length<<" "<<width<<endl; }
private:
    int length, width;
};

int main()
{ Rectangle p1(10, 20);                 // 定义对象 p1, 调用了普通的构造函数
  Rectangle p2(p1);                     // 调用拷贝构造函数, 用对象 p1 初始化对象 p2
  p1.disp();
  p2.disp();
  return 0;
}
```

本例在定义对象 p2 时, 调用了自定义拷贝构造函数。程序运行结果如下:

```
Using normal constructor
Using copy constructor
10 20
20 40
```

从运行结果可以看出, 该程序中调用过一次普通的构造函数, 用来初始化对象 p1。程序中又调用了一次拷贝构造函数, 用来将对象 p1 去初始化对象 p2。

在程序中, 用一个对象去初始化另一个对象时, 或者说, 用一个对象去复制另一个对象时, 可以有选择、有变化地复制, 类似于用复印机复制文件一样, 可大可小, 也可以复印其中的一部分。本例将对象 p1 的值逐域乘上 2 后, 复制给对象 p2。

调用拷贝构造函数的一般形式为:

类名 对象 2(对象 1);

例如上例中的

```
Rectangle p2(p1);
```

这种调用拷贝构造函数的方法称为“代入”法。除了用“代入”法调用拷贝构造函数外, 还可以采用“赋值”法调用拷贝构造函数。这种调用方法的一般形式如下:

类名 对象 2=对象 1;

例如，将例 3.15 中主函数 main 改成如下形式：

```
int main()
{ Rectangle p1(10, 20);    //定义对象 p1，调用了普通的构造函数
  Rectangle p2=p1;         //以“赋值”法调用拷贝构造函数，用对象 p1 初始化对象 p2
  p1.disp();
  p2.disp();
  return 0;
}
```

在执行语句“Rectangle p2=p1;”时，虽然从形式上看是将对象 p1 赋值给了对象 p2，但实际上调用的是拷贝构造函数，运行结果没有发生变化。

2. 默认的拷贝构造函数

每个类都必须有一个拷贝构造函数。程序员可以自定义拷贝构造函数，用于按照需要初始化新的对象。如果程序员没有定义拷贝构造函数，系统就会自动生成产生一个默认的拷贝构造函数，用于复制出与源对象的数据成员的值完全相同的新对象。

若把例 3.15 中的自定义拷贝构造函数去掉，改变为例 3.16，将调用默认的拷贝构造函数。

例 3.16 调用默认的拷贝构造函数。

```
#include<iostream>
using namespace std;
class Rectangle{
public:
  Rectangle(int len, int wid)    // 构造函数
  { length=len;
    width=wid;
    cout<<"Using normal constructor\n";
  }
  void disp()
  { cout<<length<<" " <<width<<endl; }
private:
  int length, width;
};

int main()
{ Rectangle p1(10, 20);          //定义对象 p1，调用了普通构造函数初始化对象 p1
  Rectangle p2(p1);              //以“代入”法调用默认的拷贝构造函数，用
                                //对象 p1 初始化对象 p2
  Rectangle p3=p1;               //以“赋值”法调用默认的拷贝构造函数，用
                                //对象 p1 初始化对象 p3

  p1.disp();
  p2.disp();
  p3.disp();
  return 0;
}
```

由于上例没有用户自定义的拷贝构造函数，因此在定义对象 p2 时，采用了“Rectangle p2(p1)”的形式后，用“代入”法调用了系统默认的拷贝构造函数。默认的拷贝构造函数将对象 p1 的各个域的值都拷贝给了对象 p2 相应的域，因此 p2 对象的数据成员的值与 p1 对象完全相同。在定义对象 p3 时，采用了“Rectangle p3=p1”的形式后，用“赋值”法调用了系统默认的拷贝构造函数，p1 的值逐域拷贝给对象 p3。

程序运行的结果如下：

Using normal constructo

```
10    20
10    20
10    20
```

3. 调用拷贝构造函数的 3 种情况

普通的构造函数是在对象创建时被调用，而拷贝构造函数在以下 3 种情况下都会被调用：

(1) 当用类的一个对象去初始化该类的另一个对象时。如例 3.16 主函数 main 中的下述语句：

```
Rectangle p2(p1);           // 用对象 p1 初始化对象 p2， 拷贝构造函数被调用 (代入法)
Rectangle p3=p1;            // 用对象 p1 初始化对象 p3， 拷贝构造函数被调用 (赋值法)
```

便是属于这一种情况，这时需要调用拷贝构造函数。

(2) 当函数的形参是类的对象，调用函数，进行形参和实参结合时。例如：

```
fun1(Rectangle p)           //形参是类 Rectangle 的对象 p
{ p.disp();
}
int main()
{ Rectangle p1(10, 20);
  fun1(p1);                  //调用函数 fun1 时，实参 p1 是类 Rectangle 的对象，将
                             //调用拷贝构造函数， 初始化形参对象 p

  return 0;
}
```

在 main 函数内，执行语句“fun1(p1);”便是这种情况。在调用这个函数时，对象 p1 是实参，用它来初始化被调用函数的形参 p 时，需要调用拷贝构造函数。

(3) 当函数的返回值是对象，函数执行完成，返回调用者时。例如：

```
Rectangle fun2()
{ Rectangle p1(10, 30);
  return p1;                 // 函数的返回值是对象
}
int main()
{ Rectangle p2;
  p2=fun2();                 // 函数执行完成，返回调用者时，调用拷贝构造函数
  return 0;
}
```

在函数 fun2 内，执行语句“return p1;”时，将会调用拷贝构造函数将 p1 的值拷贝到一个无名对象中，这个无名对象是编译系统在主程序中临时创建的。函数运行结束时对象 p1 消失，但临时对象会存在于语句“p2=fun2()”中。执行完这个语句后，临时无名对象的使命也就完成了，该临时对象便自动消失了。

例 3.17 调用拷贝构造函数的 3 种情况。

```
#include<iostream>
using namespace std;
class Rectangle{
public:
```

```

    Rectangle(int len=10, int wid=10);           //声明构造函数
    Rectangle(const Rectangle &p);               //声明拷贝构造函数
    void disp()
    { cout<<"length<<" "<<width<<endl; }
private:
    int length, width;
};
Rectangle::Rectangle(int len, int wid)           //定义构造函数
{ length=len;
  width=wid;
  cout<<"Using normal constructor\n";
}
Rectangle::Rectangle(const Rectangle &p)         //定义拷贝构造函数
{ length=2*p.length;
  width=2*p.width;
  cout<<"Using copy constructor\n";
}
void fun1(Rectangle p)                           //形参是类对象的函数
{ p.disp();
}
Rectangle fun2()                                 //返回值是对象的函数
{ Rectangle p4(10, 30);
  return p4;
}
int main()
{ Rectangle p1(30, 40); //定义对象 p1, 第 1 次调用普通的构造函数
  p1.disp();
  Rectangle p2(p1);     //第 1 次调用拷贝构造函数, 用对象 p1 初始化对象 p2 (情况 1)
  p2.disp();
  Rectangle p3=p1;       //第 2 次调用拷贝构造函数, 用对象 p1 初始化对象 p3 (情况 1)
  p3.disp();
  fun1(p1);              //对象 p1 作为函数 fun1 的实参
                          //第 3 次调用拷贝构造函数 (情况 2)
  p2=fun2();             //在函数 fun2 内部定义对象时, 第 2 次调用普通的构造函数
                          //函数的返回值是对象, 第 4 次调用拷贝构造函数 (情况 3)
  p2.disp();
  return 0;
}

```

程序运行结果如下:

```

Using normal constructor
30 40
Using copy constructor
60 80
Using copy constructor
60 80
Using copy constructor
60 80

```


Using normal constructor

Using copy constructor

20 60

3.6 自引用指针 this

当定义了一个类的若干对象后，系统会为每一个对象分配存储空间。如果一个类包含了数据成员和成员函数，就要分别为数据和函数的代码分配存储空间。按照通常的思路，如果一个类定义了3个对象，那么就应该分别为这3个对象的数据和函数代码分配存储空间。

事实上，给对象赋值就是给对象的数据成员赋值，不同对象的存储单元中存放的数据值通常是不相同的，而不同对象的函数代码是相同的，不论调用哪一个对象的成员函数，其实调用的都是相同内容的代码。C++的编译系统只用了一段空间来存放这个共同的函数代码段，在调用各对象的成员函数时，都去调用这个公用的函数代码。因此，每个对象的存储空间都只是该对象数据成员所占用的存储空间，而不包括成员函数代码所占用的空间，函数代码是存储在对象空间之外的。

每个对象都有属于自己的数据成员，但是所有对象的成员函数代码却合用一份。那么成员函数是怎样辨别出当前调用自己的是哪个对象，从而对该对象的数据成员而不是对其他对象的数据成员进行处理呢？下面，让我们看一个简单的例子。

例 3.18 this 指针的引例。

```
#include<iostream>
using namespace std;
class A{
public:
    A(int x1)
    { x=x1; }
    void disp()
    { cout<<"x= "<<x<<endl;}
private:
    int x;
};
int main()
{ A a(1), b(2);
  cout<<" a: ";
  a.disp();
  cout<<" b: ";
  b.disp();
  return 0;
}
```

读者不难知道，运行这个程序的结果如下：

```
a: x=1
```

```
b: x=2
```

但是，不论对象 a 还是对象 b 调用函数 disp 时都执行同一条语句

```
cout<<"x= "<<x<<endl;
```

那么,执行 `a.disp()` 时,成员函数 `disp` 怎样知道现在输出的应该是对象 `a` 的 `x` 值 1 呢? 类似的,执行 `b.disp()` 时,成员函数 `disp` 又怎样知道现在输出的应该是对象 `b` 的 `x` 值 2 呢?

原来, C++ 为成员函数提供了一个名字为 `this` 的指针, 这个指针称为自引用指针。每当创建一个对象时, 系统就把 `this` 指针初始化为指向该对象, 即 `this` 指针的值是当前调用成员函数的对象的起始地址。每当调用一个成员函数时, 系统就自动把 `this` 指针作为一个隐含的参数传给该函数。不同的对象调用同一个成员函数时, C++ 编译器将根据成员函数的 `this` 指针所指向的对象来确定应该引用哪一个对象的数据成员。因此, 被存取的必然是指定对象的数据成员, 绝不会搞错。例如, 当调用成员函数 `a.disp()` 时, 编译系统就把对象 `a` 的起始地址赋给 `this` 指针, 于是在成员函数引用数据成员时, 就按照 `this` 的指向找到对象 `a` 的数据成员。例如, `disp` 函数要输出数据成员 `x` 的值, 实际上是执行:

```
cout<<"x=" <<this->x<<endl;
```

由于当前 `this` 指针指向对象 `a`, 因此相当于执行:

```
cout<<"x=" <<a.x<<endl;
```

这样就输出了对象 `a` 的数据成员 `x` 的值。同样当调用成员函数 `b.disp()` 时, 编译系统就把对象 `b` 的起始地址赋给 `this` 指针, 于是在成员函数 `disp` 引用数据成员时, 就按照 `this` 的指向找到对象 `b` 的数据成员。此时的 “`cout<<"x=" <<this->x<<endl`” 就是 “`cout<<"x=" <<b.x<<endl`”, 显然这就输出了对象 `b` 的数据成员 `x` 的值。

下面的例子, 通过显示 `this` 的值, 我们可更加清楚地了解其功能和原理。

例 3.19 显示 `this` 指针的值。

```
#include<iostream>
using namespace std;
class A{
public:
    A(int x1)
    { x=x1; }
    void disp()
    { cout <<"\nthis="<<this<<" when x="<<this->x; }
private:
    int x;
};
int main()
{ A a(1), b(2), c(3);
  a.disp();
  b.disp();
  c.disp();
  return 0;
}
```

运行这个程序, 屏幕上显示的结果如下:

```
this=0012FF7C when x=1
this=0012FF78 when x=2
this=0012FF74 when x=3
```

分析这个例子, 我们可以看出, `this` 指针的值是随着对象的不同而改变的。

3.7 C++的 string 类

C++支持两种类型的字符串，第1种是C语言中介绍过的包括一个结束符‘\0’（即以NULL结束）的字符数组，标准库函数提供了一组对其进行操作的函数，可以完成许多常用的字符串操作，如字符串复制函数 strcpy、字符串连接函数 strcat 和求字符串长度函数 strlen 等。C++中仍保留了这种格式字符串。使用数组来存放字符串，调用标准库函数来处理字符串，使用起来不太方便，而且数据与处理数据的函数分离也不符合面向对象方法的要求。为此，在C++的标准库中，声明了另一种更方便的字符串类型，即字符串类 string，类 string 提供了对字符串进行处理所需要的操作。

使用 string 类必须在程序的开始包括头文件 string，即要有如下语句：

```
#include <string>
```

string 类的字符串对象的使用方法与其他对象一样，也必须先定义才可以使用。其定义格式如下：

string 对象 1，对象 2，…；

例如：

```
string str1, str2;           //定义 string 类对象 str1 和 str2
string str3("China");       //定义 string 类对象 str3 同时对其初始化
```

字符串对象初始化方式也可写成：

```
string str4="China";        //定义 string 类对象 str4 同时对其初始化
```

同时，C++还为 string 类的对象定义了许多应用于字符串的运算符，常用的字符串运算符如表 3-1 所示（假设表中的 s1 和 s2 均已定义为 string 类对象）。

表 3-1 常用的 string 类运算符

运算符	示 例	注 释
=	s1=s2	用 s2 给 s1 赋值
+	s1+s2	用 s1 和 s2 连接成一个新串
+=	s1+=s2	等价于 s1=s1+s2
==	s1==s2	判断 s1、s2 是否相等
!=	s1!=s2	判断 s1、s2 是否不等
<	s1<s2	判断 s1 是否小于 s2
<=	s1<=s2	判断 s1 是否小于或等于 s2
>	s1>s2	判断 s1 是否大于 s2
>=	s1>=s2	判断 s1 是否大于或等于 s2
[]	s1[i]	访问串对象 s1 中下标为 i 的字符
>>	cin>>s1	从键盘输入一个字符串给串对象 s1
<<	cout<<s1	将串对象 s1 输出

这些运算符允许在一般的表达式中使用 `string` 类对象，而不再需要调用诸如 `strcpy` 或 `strcat` 之类的函数。同时，也可以在表达式中把 `string` 类对象和以 ‘\0’ 结束的字符串混在一起使用，如可以把一个以 ‘\0’ 结束的字符串赋给一个 `string` 类对象。

C++ 的 `string` 类使得字符串的处理比使用字符串函数更直观和方便，下面举例说明这些操作。

例 3.20 `string` 类运算符的操作。

```
#include<iostream>
#include <string>
using namespace std;
int main()
{ string str1="ABC";           //定义 string 类对象 str1 并进行初始化
  string str2="DEF";           //定义 string 类对象 str2 并进行初始化
  string str3("GHI");          //定义 string 类对象 str3 并进行初始化
  string str4, str5;           //定义 string 类对象 str4 和 str5
  str4=str1;                   //字符串赋值
  cout<<"str4 is "<<str4<<endl; //字符串输出
  str5=str1+str2;              //字符串连接
  cout<<"str1+str2 is "<<str5<<endl;
  str5=str1+"123";             //字符串连接
  cout<<"str1+\\"123\\" is "<<str5<<endl;
  if (str3>str1)                //字符串比较
    cout<<"str3>str1 " <<endl;
  else cout<<"str3<str1 " <<endl;
  if (str4==str1)               //字符串比较
    cout<<"str4==str1 " <<endl;
  else cout<<"str4<>str1 " <<endl;
  cout<<"请输入一个字符串给 str5: ";
  cin>>str5;                   //从键盘输入一个字符串给 str5
  cout<<"str5 is "<<str5<<endl;
  return 0;
}
```

程序运行结果为：

```
str4 is ABC
str1+str2 is ABCDEF
str1+"123" is ABC123
str3>str1
str4==str1
```

请输入一个字符串给 str5: XYZ↵

```
str5 is XYZ
```

这个程序是很好理解的。从这个程序可以看出，使用 `string` 类后，字符串运算变得非常简单、直观。

3.8 应用举例

例 3.21 定义一个矩形 (Rectangle) 类, 私有数据成员为矩形的长度 (len) 和宽度 (wid), 无参构造函数置 len 和 wid 为 0, 有参构造函数置 len 和 wid 为对应形参的值, 有能够放大长度和宽度一倍的拷贝构造函数, 还包括求矩形周长、求矩形面积、取矩形长度和宽度的公有成员函数, 另外还有一个能够输出矩形、长度、宽度、周长和面积等公有成员函数。

```
#include<iostream>
using namespace std;
const double PI=3.14;
class Rectangle{
public:
    Rectangle()                //无参构造函数
    { len=0;
      wid=0;
    }
    Rectangle(double x, double y) //有参构造函数
    { len=x;
      wid=y;
    }
    Rectangle(const Rectangle& r) //拷贝构造函数
    { len=2*r.len;
      wid=2*r.wid;
    }
    double perimeter()          //成员函数, 求矩形周长
    { return 2*(len+wid);
    }
    double area()                //成员函数, 求矩形面积
    { return len*wid;
    }
    double getlenth()           //成员函数, 取矩形长度
    { return len;
    }
    double getwidth()           //成员函数, 取矩形宽度
    { return wid;
    }
    void display();              //成员函数, 输出矩形的长度、宽度、周长和面积
private:
    double len, wid;
};
void Rectangle::display()
{ cout<<"矩形的长是:"<<len<<"", "<<"矩形的宽是:"<<wid<<endl;
  cout<<"矩形的周长是:"<<perimeter()<<"",
  cout<<"矩形的面积是:"<<area()<<endl;
}
int main()
{ Rectangle r1(3, 6), r2;      //调用无参构造函数和有参构造函数,
                                //定义对象 r1 和 r2
```

```

    cout<<"矩形 r1 的相关信息是:"<<endl;
    r1.display();
    cout<<"*****"<<endl;
    r2=r1;           //将对象 r1 的值赋给对象 r2
    cout<<"矩形 r2 的相关信息是:"<<endl;
    r2.display();
    cout<<"*****"<<endl;
    Rectangle r3(r1); //调用拷贝构造函数,定义对象 r3
    cout<<"矩形 r3 的相关信息是:"<<endl;
    r3.display();
    return 0;
}

```

程序运行结果如下:

```

矩形 r1 的相关信息是:
矩形的长是:3, 矩形的宽是:6
矩形的周长是:18, 矩形的面积是:18
*****
矩形 r2 的相关信息是:
矩形的长是:3, 矩形的宽是:6
矩形的周长是:18, 矩形的面积是:18
*****
矩形 r3 的相关信息是:
矩形的长是:6, 矩形的宽是:12
矩形的周长是:36, 矩形的面积是:72

```

实 验

实验目的和要求

1. 理解类和对象的概念, 学习声明类和定义对象的方法。
2. 学习使用类和对象编制 C++ 程序。
3. 学习构造函数和析构函数的实现方法。
4. 学习 string 类的使用方法。

实验内容和步骤

1. 输入下列程序。

```

//test3_1.cpp
#include<iostream>
using namespace std;
class Coordinate
{ public:
    Coordinate(int x1, int y1)
    { x=x1;
      y=y1;
    }
}

```

```

Coordinate(Coordinate &p);
~Coordinate()
{ cout<<"Destructor is called\n";}
int getX()
{ return x; }
int getY()
{ return y; }
private:
int x, y;
};
Coordinate::Coordinate(Coordinate &p)
{ x=p.x;
  y=p.y;
  cout<<"Copy-initialization Constructor is called\n";
}
int main()
{ Coordinate p1(3, 4);
  Coordinate p2(p1);
  Coordinate p3=p2;
  cout<<"p3=("<<p3.getX()<<"", "<<p3.getY()<<"")\n";
  return 0;
}

```

(1) 写出程序的运行结果。

(2) 将 `Coordinate` 类中带有两个参数的构造函数进行修改，在函数体内增添下述语句：

```
cout<<"Constructor is Called. \n";
```

写出程序的运行结果，并解释输出结果。

(3) 按下列要求进行调试：

在主函数体内，添加下述语句：

```
Coordinate p4;
Coordinate p5(2);
```

调试程序时会出现什么错误？为什么？如何对已有的构造函数进行适当修改？

(4) 经过以上第(2)步和第(3)步的修改后，结合运行结果分析：创建不同的对象时会调用不同的构造函数。

2. 设计一个 4×4 魔方程序，让魔方的各行值的和等于各列值的和，并且等于两对角线值的和。例如，以下魔方：

```

31   3   5  25
 9  21  19  15
17  13  11  23
 7  27  29   1

```

各行、各列以及两对角线值的和都是 64。

【提示】

求 4×4 魔方的一般步骤如下。

(1) 设置初始魔方的起始值和相邻元素之间的差值。例如，上述魔方的初始魔方起始值 (first) 和相邻元素之间的差值 (step) 分别为：

```
first=1
```

step=2

(2) 设置初始魔方元素的值。例如, 上述魔方的初始魔方为:

```
1   3   5   7
9   11  13  15
17  19  21  23
25  27  29  31
```

(3) 生成最终魔方。方法如下。

① 求最大元素值与最小元素值的和 sum, 该实例的 sum 是:

$1+31=32$

② 用 32 减去初始魔方所有对角线上元素的值, 然后将结果放在原来的位置, 这样就可求得最终魔方。本例最终魔方如下:

```
31   3   5   25
9   21  19  15
17  13  11  23
7   27  29   1
```

本题魔方类 magic 的参考框架如下:

```
class magic                                //声明魔方类 magic
{ public:
    void getdata();                        //输入初值成员函数
    void setfirstmagic();                  //设置初始魔方成员函数
    void generatemagic();                  //生成最终魔方成员函数
    void printmagic();                     //显示魔方成员函数
private:
    int m[4][4];
    int step;                             //相邻元素之间的差值
    int first;                             //起始值
    int sum;                               //最大元素值和最小元素值的和
};
```

3. 使用 C++ 的 string 类, 将 5 个字符串按逆转后的顺序显示出来。例如, 逆转前的 5 个字符串是:

Germany Japan America Britain France

逆转后的顺序输出字符串是:

France Britain America Japan Germany

习 题

【3.1】在不考虑保护成员的情况下, 类声明的一般格式是什么?

【3.2】下列关于类和对象的描述中, 不正确的是 ()。

- A. 一个对象只能属于一个类
- B. 对象是类的一个实例
- C. 一个类只能有一个对象
- D. 类和对象的关系与数据类型和变量的关系类似

【3.3】若有如下类声明:

```
class A {
    int a;
};
```

则类 A 的成员 a 是 ()。

- A. 公有数据成员
- B. 私有数据成员
- C. 公有成员函数
- D. 私有成员函数

【3.4】假定有一个类,类名为 Date,则执行语句“Date d;”时将自动调用该类的 ()。

- A. 有参构造函数
- B. 无参构造函数
- C. 拷贝构造函数
- D. 赋值重载函数

【3.5】下列关于构造函数的描述中,错误的是 ()。

- A. 构造函数可以设置默认参数
- B. 构造函数在定义类的对象时自动执行
- C. 构造函数可以是内联函数
- D. 构造函数不可以重载

【3.6】在下面有关对构造函数的描述中,正确的是 ()。

- A. 构造函数可以带有返回值
- B. 构造函数的名字与类名完全相同
- C. 构造函数必须带有参数
- D. 构造函数必须定义,不能默认

【3.7】构造函数是在 () 时被执行的。

- A. 程序编译
- B. 创建对象
- C. 创建类
- D. 程序装入内存

【3.8】在类外定义成员函数时,需要在函数名前加上 ()。

- A. 对象名
- B. 类名
- C. 类名和作用域运算符
- D. 作用域运算符

【3.9】类的析构函数是在 () 调用的。

- A. 创建类时
- B. 创建对象时
- C. 撤销对象时
- D. 访问对象成员时

【3.10】假定有一个类,类名为 Date,在下面构造函数的原型声明中存在着语法错误的是 ()。

- A. Date(int a, int);
- B. int Date(int, int);
- C. Date(int, int);
- D. Date(int, int y);

【3.11】在下面有关析构函数特征的描述中,正确的是 ()。

- A. 一个类中可以定义多个析构函数
- B. 析构函数名与类名完全相同
- C. 析构函数不能指定返回类型
- D. 析构函数可以有一个或多个参数

【3.12】假定有一个类,类名为 Date,则该类拷贝构造函数的原型为 ()。

- A. Date&(Date x);
- B. Date(Date x);
- C. Date(Date &x);
- D. Date(Date* x);

【3.13】通常拷贝构造函数的参数是 ()。

- A. 某个对象名
- B. 某个对象的成员名
- C. 某个对象的引用名
- D. 某个对象的指针变量名

【3.14】关于 this 指针的以下说法,正确的是 ()。

- A. this 指针必须显式说明
- B. 当创建一个对象后, this 指针就指向该对象
- C. this 指针指向成员函数

D. this 指针的值不会随着对象的不同而改变

【3.15】使用 string 类时, 必须在程序的开始包括头文件 ()。

A. cstring B. cmath C. cstdio D. string

【3.16】假设在程序中已经声明了类 point, 并建立了其对象 p1 和 p4。请回答以下几个语句有什么区别?

(1) point p2, p3;

(2) point p2=p1;

(3) point p2(p1);

(4) p4=p1;

【3.17】写出下面程序的运行结果。

```
#include<iostream>
using namespace std;
class Exam {
public:
    Exam(int x) //构造函数
    { num=x;
    }
    ~Exam() //析构函数
    { cout<<"dst "<<num<<endl;
    }
private:
    int num;
};
int main()
{ Exam w(55);
  cout<<"Exit main"<<endl;
  return 0;
}
```

【3.18】写出下面程序的运行结果。

```
#include<iostream>
using namespace std;
class test
{ public:
    test();
    ~test(){ };
private:
    int i;
};
test::test()
{ i = 25;
  for (int ctr=0; ctr<10; ctr++)
  { cout<<"Counting at "<<ctr<<"\n"; }
}
test anObject;
int main()
{ return 0;
}
```

【3.19】写出下面程序的运行结果。

```
#include<iostream>
using namespace std;
class Test{
private:
    int val;
public:
    Test()
    { cout<<"default."<<endl; }
    Test(int n)
    { val=n;
      cout<<"Con."<<endl;
    }
    Test(const Test& t)
    { val=t.val;
      cout<<"Copy con."<<endl;
    }
};

int main()
{ Test t1(6);
  Test t2=t1;
  Test t3;
  t3=t1;
  return 0;
}
```

【3.20】指出下列程序中的错误，并说明为什么？

```
#include<iostream>
using namespace std;
class Student{
    int sno;
    int age;
    void printStu();
    void setSno(int d);
};

void printStu()
{ cout<<"\nSno is "<<sno<<" ";
  cout<<"age is "<<age<<"."<<endl;
}

void setSno(int s)
{ sno=s; }
void setAge(int a)
{ age=a; }

int main()
{ Student lin;
  lin.setSno(20021);
  lin.setAge(20);
  lin.printStu();
}
```

【3.21】指出下列程序中的错误，并说明为什么？

```
#include<iostream>
using namespace std;
class Point{
```

```

public:
    int x, y;
private:
    Point()
    { x=1; y=2;
    }
};

int main()
{ Point cpoint;
  cpoint.x=2;
  return 0;
}

```

【3.22】声明一个 Circle 类，有数据成员 radius（半径）、成员函数 area()，计算圆的面积，构造一个 Circle 的对象进行测试。

【3.23】建立类 cylinder，cylinder 的构造函数被传递了两个 double 值，分别表示圆柱体的半径和高度。用类 cylinder 计算圆柱体的体积，并存储在一个 double 变量中。在类 cylinder 中包含一个成员函数 vol，用来显示每个 cylinder 对象的体积。

【3.24】定义一个日期类 Date，该类对象存放一个日期，可以提供的操作有：

```

void :printDate();           //显示日期，格式如“日期是:2010年6月8日”
void GetYear();             //取年的值
void GetMonth();            //取月的值
void GetDay();              //取日的值
void SetDate(int Y, int m, int d), //设置日期值

```

还允许对日期对象作以下定义：

```

Date d1(2010, 6, 8);        //用所给日期定义一个日期变量
Date d2;                    //定义一个日期对象
Date d3(d1);                 //用已有的日期构造一个新对象

```

要求每一个成员函数都要被调用。

第 4 章

类和对象的进一步讨论

本章进一步对类和对象其他方面的内容进行讨论，这些内容包括对象数组与对象指针、向函数传递对象的方法、静态成员、友元，以及类的组合和共享数据保护的方法。此外，对多文件程序也作了介绍。本章将通过一些例子进一步熟悉类和对象在编程中的应用，从而进一步理解类和对象的作用。

4.1 对象数组与对象指针

4.1.1 对象数组

所谓对象数组是指每一个数组元素都是对象的数组，也就是说，若一个类有若干个对象，我们把这一系列的对象用一个数组来存放。对象数组的元素是对象，不仅具有数据成员，而且还具有函数成员。

定义一个一维对象数组的格式如下：

类名 数组名[下标表达式];

假如有 5 个矩形，每个矩形的属性包括长度与宽度。如果为每一个矩形建立一个对象，需要分别取 5 个对象名。显然用程序处理起来很不方便。这时可以定义一个矩形类 `Rectangle` 的对象数组，每一个数组元素是 `Rectangle` 类的一个对象，例如：

```
Rectangle rec[5];    //定义类 Rectangle 的对象数组 rec，含有 5 个对象数组元素
```

在建立数组时，同样要调用构造函数。有几个数组元素就要调用几次构造函数。例如，有 5 个数组元素，就要调用 5 次构造函数。类 `Rectangle` 的构造函数有两个参数，分别用于给长度数据和宽度数据赋值。在介绍类 `Rectangle` 对象数组的初始化之前，我们先看一个只有一个参数的构造函数例子。如果构造函数只有一个参数，在定义对象数组时可以直接在等号后面的花括号内提供实参。

例 4.1 用只有一个参数的构造函数给对象数组赋值。

```
#include<iostream>
using namespace std;
class exam{
public:
    exam(int n)                //只有一个参数的构造函数
    { x=n; }
```

```

        int get_x()
        { return x; }
    private:
        int x;
};
int main()
{ exam obl[4]={11, 22, 33, 44};          //用只有一个参数的构造函数给对象数组赋值
  for (int i=0;i<4;i++)
    cout<<obl[i].get_x()<<' ';
  return 0;
}

```

本例在执行语句“exam obl[4]={11, 22, 33, 44};”时，定义了类 exam 的一个对象数组 obl，其含有 4 个对象数组元素，定义时共 4 次调用带参数的构造函数，分别用实参 11、22、33 和 44 初始化对象数组元素 obl[0]、obl[1]、obl[2]和 obl[3]的数据成员 x。

与基本数据类型的数组一样，在使用对象数组时也只能访问单个数组元素，其一般形式是：

数组名[下标].成员名

本例在执行语句：

```

for (int i=0;i<4;i++)
    cout<<obl[i].get_x()<<' ';

```

时，相当于执行了以下 4 条语句：

```

cout<<obl[0].get_x()<<' ';
cout<<obl[1].get_x()<<' ';
cout<<obl[2].get_x()<<' ';
cout<<obl[3].get_x()<<' ';

```

程序运行结果如下：

```
11 22 33 44
```

在设计类的构造函数时就要充分考虑到对象数组元素初始化的需要。当各个元素的初始值为相同的值时，可以在类中定义不带参数的构造函数或带有默认参数值的构造函数；当各元素对象的初值要求为不同的值时，需要定义带参数的构造函数。请看下面的例子。

例 4.2 用不带参数和带一个参数的构造函数给对象数组赋值。

```

#include<iostream>
using namespace std;
class exam{
public:
    exam()          //不带参数的构造函数
    { x=123; }
    exam(int n)     //带 1 个参数的构造函数
    { x=n; }
    int get_x()
    { return x; }
private:
    int x;
};
int main()
{ exam obl[4]={ 11, 22, 33, 44 };

```

```

exam ob2[4]={ 55, 66 };
for (int i=0;i<4;i++)
    cout<<ob1[i].get_x()<<' ';
cout<<endl;
for (i=0;i<4;i++)
    cout<<ob2[i].get_x()<<' ';
return 0;
}

```

程序运行结果如下：

```
11 22 33 44
```

```
55 66 123 123
```

说明：

本例在执行语句“exam ob1[4]={11, 22, 33, 44};”时，先后4次调用带1个参数的构造函数，分别初始化ob1[0]、ob1[1]、ob1[2]和ob1[3]。如果没有指定初始值，就调用不带参数的构造函数，例如：

```
exam ob2[4]={55, 66};
```

在执行时，首先调用带参数的构造函数，初始化ob2[0]和ob2[1]，然后调用不带参数的构造函数，初始化ob2[2]和ob2[3]。

在本例中，编译系统只为对象数组元素的构造函数传递一个实参，所以在定义数组时提供的实参个数不能超过数组元素个数，例如：

```
exam ob1[4]={11, 22, 33, 44, 55}; //编译出错，实参个数超过对象数组元素个数
```

以上例子中构造函数只有一个参数，如果构造函数有多个参数，在定义对象数组应当怎样实现初始化呢？我们只需在花括中分别写出构造函数并指定实参即可。例如，类Rectangle的构造函数有两个参数，分别代表复数的实部和虚部，则可以这样定义对象数组：

```

Rectangle rec[3]={           //定义对象数组rec
    Rectangle(10, 20),        //调用构造函数，为第1个对象数组元素提供实参10和20
    Rectangle(330, 40),       //调用构造函数，为第2个对象数组元素提供实参30和40
    Rectangle(50, 60)         //调用构造函数，为第3个对象数组元素提供实参50和60
};

```

由于这个对象数组有3个对象数组元素，因此在建立对象数组时，3次调用构造函数，对每一个对象数组元素初始化。每一个元素的实参分别用括号包起来，对应构造函数的一组实参，不会产生混淆。

例4.3 用有多个参数的构造函数给对象数组赋值。

```

#include<iostream>
using namespace std;
class Rectangle{
public:
    Rectangle(int len=10, int wid=10)        //在定义构造函数时指定默认参数值
    { length=len; width=wid; }
    int area()
    { return (length*width); }
private:
    int length, width;

```

```

};
int main()
{
    Rectangle rec[3]={
        Rectangle(10, 20), //定义对象数组
        Rectangle(30, 40), //调用构造函数, 为第1个元素 rec[0] 提供实参 10 和 20
        Rectangle(50, 60) //调用构造函数, 为第2个元素 rec[1] 提供实参 30 和 40
    };
    cout<<"The area of rec[0] is "<<rec[0].area()<<endl; //调用构造函数, 为第3个元素 rec[2] 提供实参 50 和 60
    //调用 rec[0] 的 area 函数
    cout<<"The area of rec[1] is "<<rec[1].area()<<endl;
    //调用 rec[1] 的 area 函数
    cout<<"The area of rec[2] is "<<rec[2].area()<<endl;
    //调用 rec[2] 的 area 函数
    return 0;
}

```

程序运行结果如下:

The area of rec[0] is 200

The area of rec[1] is 1200

The area of rec[2] is 3000

4.1.2 对象指针

每一个对象在初始化后都会在内存中占有一定的空间。因此,既可以通过对象名访问一个对象,也可以通过对象地址来访问一个对象。对象指针就是用于存放对象地址的变量。声明对象指针的一般语法形式为:

类名* 对象指针名;

1. 用对象指针访问单个对象成员

说明对象指针的语法和说明其他数据类型指针的语法相同。使用对象指针时,首先要把它指向一个已创建的对象,然后才能访问该对象的公有成员。

在一般情况下,用点运算符(.)来访问对象的成员,当用指向对象的指针来访问对象成员时,就要用“>”操作符。下例说明了对对象指针的使用。

例 4.4 对象指针的使用。

```

#include<iostream>
using namespace std;
class Rectangle{
public:
    void setRec(int len, int wid)
    { length=len;
      width=wid;
    }
    void disp()
    { cout<<length<<" "<<width<<endl; }
private:
    int length, width;
};
int main()
{
    Rectangle rec ; // 定义类 Rectangle 的对象 rec
}

```



```

Rectangle *pr;           // 定义 pr 为指向类 Rectangle 的对象指针变量
rec.setRec(20, 30);      // 调用对象 rec 中的函数 setRec
pr=&rec;                 // 将对象 rec 的起始地址赋给 pr
pr->disp();              // 调用 pr 所指向的对象 rec 中的函数 disp
return 0;
}

```

在这个例子中，声明了一个类 `Rectangle`，`rec` 是类 `Rectangle` 的一个对象，`pr` 是指向类 `Rectangle` 的对象指针变量，对象 `rec` 的地址是用地址操作符（&）获得并赋给对象指针变量 `pr` 的。语句“`pr->disp()`；”表示调用 `pr` 所指向的对象 `rec` 中的函数 `disp`，即等价于语句“`rec.disp()`；”。

程序的运行结果如下：

```
20 30
```

2. 用对象指针访问对象数组

对象指针不仅能访问单个对象，也能访问对象数组。下面的语句声明了一个对象指针和一个有两个元素的对象数组：

```

Rectangle *pr;           // 定义指向类 Rectangle 的对象指针变量 pr
Rectangle rec[2];        // 定义类 Rectangle 的对象数组 rec

```

若只有数组名，则该数组名代表第 1 个数组元素的地址，所以执行语句：

```
pr=rec;
```

就把对象数组 `rec` 的第 1 个元素 `rec[0]` 的地址赋给对象指针 `pr`。例如，将例 4.3 的主函数 `main` 改写为：

```

int main()
{
    Rectangle rec[2];           // 定义类 Rectangle 的对象数组 rec
    Rectangle *pr;              // 定义 pr 为指向类 Rectangle 的对象指针变量
    rec[0].setRec(20, 30);      // 调用元素 rec[0] 的成员函数 setRec
    rec[1].setRec(40, 60);      // 调用元素 rec [1] 的成员函数 setRec
    pr=rec;                     // 将对象数组 rec 的起始地址赋给 pr
    pr-> disp();                 // 调用 pr 所指向的对象数组 rec 中第 1 个元素的
                                // 函数 disp，即 rec[0].disp()

    pr++;                       // 对象指针变量 pr 加 1
                                // 即指向下一个元素 rec[1] 的地址

    pr-> disp();                 // 调用 pr 所指向的对象 rec 数组中第 2 个元素的
                                // 函数 disp，即 rec[1].disp

    return 0;
}

```

程序运行结果如下：

```
20 30
```

```
40 60
```

一般而言，当指针加 1 或减 1 时，它总是指向其基本类型中相邻的一个元素，对象指针也是如此。本例中对对象指针 `pr` 加 1 时，指向下一个对象数组元素。

4.2 向函数传递对象

4.2.1 使用对象作为函数参数

对象可以作为参数传递给函数，其方法与传递基本数据类型的变量相同。在向函数传递对象时，是通过“传值调用”递给函数的，即单向传递，只由实参传给形参，而不能由形参传回来给实参。因此函数中对对象的任何修改均不影响调用该函数的对象（实参）本身。下例说明了这一点。

例 4.5 使用对象作为函数参数。

```
#include<iostream>
using namespace std;
class Fun_para{
public:
    Fun_para(int n)
    { i=n;}
    void set_i(int n)
    { i=n;}
    int get_i()
    { return i;}
private:
    int i;
};

void add_i(Fun_para ob)           //对象 ob 作为函数 add_i 的形参
{ ob.set_i(ob.get_i()+ob.get_i());
  cout<<"在函数 add_i 内，形参对象 ob 的数据成员 i 的值为:"<<ob.get_i();
  cout<<endl;
}

int main()
{ Fun_para obj(10);
  cout<<"调用函数 add_i 前，实参对象 obj 的数据成员 i 的值为:";
  cout<<obj.get_i()<<endl;
  add_i(obj);                     //调用函数 add_i，实参为对象 obj
  cout<<"调用函数 add_i 后，实参对象 obj 的数据成员 i 的值为:";
  cout<<obj.get_i();
  return 0;
}
```

程序运行结果如下：

调用函数 add_i 前，实参对象 obj 的数据成员 i 的值为:10

在函数 add_i 内，形参对象 ob 的数据成员 i 的值为:20

调用函数 add_i 后，实参对象 obj 的数据成员 i 的值为:10

从运行结果可以看出，本例函数 add_i 中对对象的任何修改均不影响调用该函数的对象本身。但是，我们也可以将对象的地址传递给函数，此时在函数中对形参对象的修改将影响调用该函数的实参对象本身。下面介绍有关的方法。

4.2.2 使用对象指针作为函数参数

对象指针可以作为函数的参数,使用对象指针作为函数参数可以实现传址调用,即在函数调用时使实参对象和形参对象指针变量指向同一内存地址,在函数调用过程中,形参对象指针所指对象值的改变也同样影响着实参对象的值。

当函数的形参是对象指针时,调用函数的对应实参应该是某个对象的地址值。下面我们对例 4.5 稍作修改,说明对象指针作为函数参数这个问题。

例 4.6 使用对象指针作为函数参数。

```
#include<iostream>
using namespace std;
class Fun_para{
public:
    Fun_para(int n)
    { i=n;}
    void set_i(int n)
    { i=n;}
    int get_i()
    { return i;}
private:
    int i;
};

void add_i(Fun_para *ob)           //对象指针作为函数 add_i 的形参
{ ob->set_i(ob->get_i()+ob->get_i());
  cout<<"在函数 add_i 内,形参对象 ob 的数据成员 i 的值为:"<<ob->get_i();
  cout<<endl;
}

int main()
{ Fun_para obj(10);
  cout<<"调用函数 add_i 前,实参对象 obj 的数据成员 i 的值为:";
  cout<<obj.get_i()<<endl;
  add_i(&obj);                    //调用函数 add_i,实参为对象 obj 的地址
  cout<<"调用函数 add_i 后,实参对象 obj 的数据成员 i 的值为:";
  cout<<obj.get_i();
  return 0;
}
```

程序运行结果如下:

调用函数 add_i 前,实参对象 obj 的数据成员 i 的值为:10

在函数 add_i 内,形参对象 ob 的数据成员 i 的值为:20

调用函数 add_i 后,实参对象 obj 的数据成员 i 的值为:20

不难看出,调用函数前实参对象 obj.i 的值是 10,函数调用中形参对象 ob.i 的值修改为 20,函数调用后实参对象 obj.i 的值也变为 20。可见形参对象指针所指对象的值的改变也同样影响着实参对象的值。

4.2.3 使用对象引用作为函数参数

在实际应用中,使用对象引用作为函数参数非常普遍,大部分程序员喜欢用对象引用取代对象指针作为函数参数。因为使用对象引用作为函数参数不但具有对象指针用作函数参数

的优点,而且用对象引用作函数参数将更简单、更直接。下面我们对例 4.6 稍作修改,说明对象引用作为函数参数这个问题。

例 4.7 使用对象引用作为函数参数。

```
#include<iostream>
using namespace std;
class Fun_para{
public:
    Fun_para(int n)
    { i=n;}
    void set_i(int n)
    { i=n;}
    int get_i()
    { return i;}
private:
    int i;
};

void add_i(Fun_para &ob)           //对象引用作为函数 add_i 的形参
{ ob.set_i(ob.get_i()+ob.get_i());
  cout<<"在函数 add_i 内,形参对象 ob 的数据成员 i 的值为:"<<ob.get_i();
  cout<<endl;
}

int main()
{ Fun_para obj(10);
  cout<<"调用函数 add_i 前,实参对象 obj 的数据成员 i 的值为:";
  cout<<obj.get_i()<<endl;
  add_i(obj);                      //调用函数 add_i,实参为对象 obj
  cout<<"调用函数 add_i 后,实参对象 obj 的数据成员 i 的值为:";
  cout<<obj.get_i();
  return 0;
}
```

程序运行结果如下:

调用函数 add_i 前,实参对象 obj 的数据成员 i 的值为:10

在函数 add_i 内,形参对象 ob 的数据成员 i 的值为:20

调用函数 add_i 后,实参对象 obj 的数据成员 i 的值为:20

说明:

本例和例 4.6 的主要区别在于例 4.6 使用对象指针作为函数参数,而本例使用对象引用作为函数参数,两个例子的输出结果是完全相同的。请读者比较一下这两种函数参数在使用上的区别。

4.3 静态成员

前面已经介绍过,如果一个类有多个对象,那么每一个对象都分别有自己的数据成员,不同对象的数据成员有各自的值,相互独立,互不相干。但是有时人们希望某一个或几个数据成员为所有的对象所共有,实现一个类的多个对象之间的数据共享,C++提出了静态成员的概念。静态成员包括静态数据成员和静态成员函数。下面分别对它们进行讨论。

4.3.1 静态数据成员

对象是类的一个实例，每个对象具有自己的数据成员。例如，学生类对象张三或李四，都具有自己的学号、姓名和成绩。在实际使用时，常常还需要一些其他的数据项，比如学生人数、总成绩和平均成绩等。但是，如果把这些数据项也作为普通的数据成员来处理，将会产生错误。下面通过例题来予以说明。

例 4.8 静态数据成员的引例。

```
#include<iostream>
#include<string>
using namespace std;
class Student {
public:
    Student(string namel, float score1);
    void show();                //声明成员函数 show 的原型
    void show_count_sum_ave();   //声明成员函数 show_count_sum_ave 的原型
private:
    string name;                //学生姓名
    float score;                //学生成绩
    int count;                  //学生人数
    float sum;                  //累加成绩
    float ave;                  //平均成绩
};
Student::Student(string namel, float score1)    //定义构造函数
{
    name=namel;
    score=score1;
    ++count;                                    //累加学生人数
    sum=sum+score;                             //累加成绩
    ave=sum/count;                             //计算平均成绩
}
void Student::show()                          //定义成员函数 show，输出姓名和成绩
{
    cout<<"姓名: "<<name<<endl;
    cout<<"成绩: "<<score<<endl;
}
void Student::show_count_sum_ave()             //定义成员函数 show_count_sum_ave
{
    cout<<"学生人数: "<<count<<endl;           //输出学生人数、累加成绩和平均成绩
    cout<<"累加成绩: "<<sum<<endl;
    cout<<"平均成绩: "<<ave<<endl;
}
int main()
{
    Student stu1("Liming", 90);                //建立第1个学生对象 stu1
    stu1.show();
    stu1.show_count_sum_ave();
    Student stu2("Zhanghao", 80);              //建立第2个学生对象 stu2
    stu2.show();
    stu2.show_count_sum_ave();
    return 0;
}
```

本例的设计思想是,希望每定义一个对象,调用一次构造函数,使数据成员 count 加 1,从而累计学生人数;同时用数据成员 sum 累加学生成绩,求出学生的总成绩,用数据成员 ave 计算平均成绩。程序的运行结果如下:

```

姓名: Liming
成绩: 90
学生人数: -858993459      (注: 此结果错误)
累加成绩: -1.07374e+008   (注: 此结果错误)
平均成绩: 0.125          (注: 此结果错误)

姓名: Zhanghao
成绩: 80
学生人数: -858993459      (注: 此结果错误)
累加成绩: -1.07374e+008   (注: 此结果错误)
平均成绩: 0.125          (注: 此结果错误)

```

不难看出,这个例题的运行结果是错误的。其原因是,一个学生对象的 count、sum 和 ave 仅仅属于这个学生对象,而不是所有学生对象所共享的,因此它们不能表示所有学生的人数、总成绩和平均成绩。

要想统计学生人数、总成绩和平均成绩, count、sum 和 ave 不能够定义为类的普通数据成员,必须使它们为所有的学生对象共享。那么,怎样才能使 count、sum 和 ave 被多个对象共享呢?

如果将 count、sum 和 ave 说明为全局变量,这样可以达到多个对象数据共享的目的。但是使用全局变量会带来不安全性,并且破坏了面向对象程序设计的信息隐蔽技术,与面向对象的封装性特征是矛盾的。为了实现同一个类的多个对象之间的数据共享, C++ 提出了静态数据成员的概念。

在一个类中,若将一个数据成员说明为 static,这种成员称为静态数据成员。与一般的数据成员不同,无论建立多少个类的对象,都只有一个静态数据成员的拷贝。从而实现了同一个类的不同对象之间的数据共享。

定义静态数据成员的格式如下:

static 数据类型 数据成员名;

下面将例 4.8 中 count、sum 和 ave 改为静态数据成员来处理,就能很好地完成预想的功能。

例 4.9 静态数据成员的使用。

```

#include<iostream>
#include<string>
using namespace std;
class Student {
public:
    Student(string name1, float score1);
    void show();                //声明成员函数 show 的原型
    void show_count_sum_ave();   //声明成员函数 show_count_sum_ave 的原型
private:
    string name;                // 学生姓名

```

```

float score;           // 学生成绩
static int count;      // 静态数据成员,用于统计学生人数
static float sum;      // 静态数据成员,用于统计累加成绩
static float ave;      // 静态数据成员,用于统计平均成绩
};

Student::Student(string name1, float score1) //定义构造函数
{
    name=name1;
    score=score1;
    ++count;           // 累加学生人数
    sum=sum+score;     // 累加成绩
    ave=sum/count;     // 计算平均成绩
}

void Student::show()   //定义成员函数 show, 输出姓名和成绩
{
    cout<<"姓名: "<<name<<endl;
    cout<<"成绩: "<<score<<endl;
}

void Student::show_count_sum_ave() //定义成员函数 show_count_sum_ave
{
    cout<<"学生人数: "<<count<<endl; //输出学生人数、累加成绩和平均成绩
    cout<<"累加成绩: "<<sum<<endl;
    cout<<"平均成绩: "<<ave<<endl;
}

int Student::count=0; // 静态数据成员 count 初始化
float Student::sum=0.0; // 静态数据成员 sum 初始化
float Student::ave=0.0; // 静态数据成员 ave 初始化

int main()
{
    Student stu1("Liming", 90); //建立第1个学生对象 stu1
    stu1.show();
    stu1.show_count_sum_ave();
    Student stu2("Zhanghao", 80); //建立第2个学生对象 stu2
    stu2.show();
    stu2.show_count_sum_ave();
    return 0;
}

```

程序运行结果如下:

姓名: Liming

成绩: 90

学生人数: 1

累加成绩: 90

平均成绩: 90

姓名: Zhanghao

成绩: 80

学生人数: 2

累加成绩: 170

平均成绩: 85

在上面的例子中,类 Student 的数据成员 count、sum 和 ave 被声明为静态的,它们为所

有 Student 类的对象所共享。因此,每定义一个对象,调用一次构造函数,使数据成员 count 加 1,从而累计学生人数;同时数据成员 sum 累加每个对象(即学生)的成绩,求出学生的累加成绩,数据成员 ave 计算出所有学生的平均成绩。不难看出,本例的运行结果是正确的。

说明:

(1) 静态数据成员的定义与普通数据成员相似,但前面要加上 static 关键字。例如;

```
string name;           // 普通数据成员,用于表示学生姓名
float score;           // 普通数据成员,用于表示学生成绩
static int count;       // 静态数据成员,用于统计学生人数
static float sum;       // 静态数据成员,用于统计累加成绩
static float ave;       // 静态数据成员,用于统计平均成绩
```

(2) 静态数据成员的初始化与普通数据成员不同。静态数据成员初始化应在类外单独进行,而且应在定义对象之前进行。一般在 main 函数之前,类声明之后的特殊地带为它提供定义和初始化。初始化的格式如下:

数据类型 类名::静态数据成员名 = 初始值;

例如,上面的静态数据成员,在定义对象之前就应该先进行如下的初始化:

```
int Student::count=0;   //前面不要加 static
float Student::sum=0.0; //前面不要加 static
float Student::ave=0.0; //前面不要加 static
```

如果没有对静态数据成员赋值,如:

```
int Student::count;
```

则编译系统会自动赋予初值 0,等价于

```
int Student::count=0;
```

(3) 静态数据成员属于类(准确地说,是属于类中对象的集合),而不像普通数据成员那样属于某一对象,因此可以使用“类名::”访问静态的数据成员。用类名访问静态数据成员的格式如下:

类名::静态数据成员名

例如,上面例子中的 Student::count 和 Student::sum 等。

(4) 静态数据成员与静态变量一样,是在编译时创建并初始化的。它在该类的任何对象被建立之前就存在。因此,公有的静态数据成员可以在对象定义之前被访问。对象定义后,公有的静态数据成员,也可以通过对象进行访问。用对象访问静态数据成员的格式如下:

对象名.静态数据成员名;

对象指针->静态数据成员名;

例 4.10 公有静态数据成员的访问。

```
#include<iostream>
using namespace std;
class myclass {
public:
    static int i;
};
```



```

int myclass::i=0; //静态数据成员初始化,不必在前面加 static
int main()
{ myclass::i=200; //公有静态数据成员可以在对象定义之前被访问
  myclass obl,*p;
  p=&obl;
  cout<<"obl.i: " <<obl.i<<endl; //通过对象进行访问公有静态数据成员 i
  cout<<"myclass::i: " <<myclass::i<<endl; //通过类名访问公有静态数据成员 i
  cout<<"p->i: " <<p->i<<endl; //通过对象指针访问公有静态数据成员 i
  return 0;
}

```

程序运行结果如下:

```

obl.i:      200
myclass::i: 200
p->i:       200

```

(5) 在类外,私有静态数据成员不能直接访问,必须通过公有的成员函数访问。

(6) C++支持静态数据成员的一个主要原因是可以不必使用全局变量。依赖于全局变量的类几乎都是违反面向对象程序设计的封装特性的。静态数据成员的主要用途是定义类的所有对象公用的数据,如统计总数、平均数等。

4.3.2 静态成员函数

在类定义中,前面有 `static` 说明的成员函数称为静态成员函数。静态成员函数属于整个类,是该类所有对象共享的成员函数,而不属于类中的某个对象。与静态成员函数不同,静态成员函数的作用不是为了对象之间的沟通,而是为了处理静态数据成员。定义静态成员函数的格式如下:

`static 返回类型 静态成员函数名(参数表);`

与静态数据成员类似,调用公有静态成员函数的一般格式有如下几种:

类名::静态成员函数名(实参表)

对象.静态成员函数名(实参表)

对象指针->静态成员函数名(实参表)

下面我们将例 4.9 稍加改动,使用静态成员函数来访问静态数据成员。

例 4.11 用静态成员函数访问静态数据成员。

```

#include<iostream>
#include<string>
using namespace std;
class Student{
private:
    string name; // 非静态数据成员,用于表示学生姓名
    float score; // 非静态数据成员,用于表示学生成绩
    static int count; // 静态数据成员,用于统计学生人数
    static float sum; // 静态数据成员,用于统计累加成绩
    static float ave; // 静态数据成员,用于统计平均成绩
public:
    Student(string name1, float score1);

```

```

        void show(); // 普通成员函数, 输出姓名、学号和成绩
    static void show_count_sum_ave(); // 静态成员函数, 输出学生
                                        // 人数和累加成绩及平均成绩
};
Student::Student(string name1, float score1)
{
    name=name1;
    score=score1;
    ++count; // 累加学生人数
    sum=sum+score; // 累加成绩
    ave=sum/count; // 计算平均成绩
}
void Student::show()
{
    cout<<"姓名: "<<name<<endl;
    cout<<"成绩: "<<score<<endl;
}
void Student::show_count_sum_ave() // 静态成员函数
{
    cout<<"学生人数: "<<count<<endl; // 输出静态数据成员 count
    cout<<"累加成绩: "<<sum<<endl; // 输出静态数据成员 sum
    cout<<"平均成绩: "<<ave<<endl; // 输出静态数据成员 ave
}
int Student::count=0; // 初始化静态数据成员 count
float Student::sum=0.0; // 初始化静态数据成员 sum
float Student::ave=0.0; // 初始化静态数据成员 ave
int main()
{
    Student stu1("Liming", 90);
    stu1.show();
    Student::show_count_sum_ave(); // 使用类名访问静态成员函数
    Student stu2("Zhanghao", 80);
    stu2.show();
    stu2.show_count_sum_ave(); // 使用对象名 stu2 访问静态成员函数
    return 0;
}

```

本例中定义了静态成员函数 `show_count_sum_ave()`, 对静态数据成员 `count`、`sum` 和 `ave` 进行操作。本例采用类名和对象名两种方法访问静态成员函数。

程序的运行结果如下:

姓名: Liming

成绩: 90

学生人数: 1

累加成绩: 90

平均成绩: 90

姓名: Zhanghao

成绩: 80

学生人数: 2

累加成绩: 170

平均成绩: 85

一般而言,静态成员函数不访问类中的非静态成员。若确实需要,静态成员函数只能通过对象名(或对象指针、对象引用)访问该对象的非静态成员。

下面的例子给出了静态成员函数访问非静态数据成员的方法。

例 4.12 用静态成员函数访问非静态数据成员。

```
#include<iostream>
#include<string>
using namespace std;
class Student{
private:
    string name;           // 非静态数据成员,用于表示学生姓名
    float score;           // 非静态数据成员,用于表示学生成绩
    static int count;       // 静态数据成员,用于统计学生人数
    static float sum;       // 静态数据成员,用于统计累加成绩
    static float ave;       // 静态数据成员,用于统计平均成绩
public:
    Student(string name1, float score1);

    static void show(Student &stu);    // 静态成员函数,输出非静态数据成员
                                        // 函数参数为对象的引用
    static void show_count_sum_ave();  // 静态成员函数,输出学生
                                        // 人数和累加成绩及平均成绩
};

Student::Student(string name1, float score1 )
{ name=name1;
  score=score1;
  ++count;           // 累加学生人数
  sum=sum+score;     // 累加成绩
  ave=sum/count;     // 计算平均成绩
}

void Student::show(Student &stu)      // 静态成员函数,输出
{ cout<<"姓名: "<<stu.name<<endl;    // 非静态数据成员 name
  cout<<"成绩: "<<stu.score<<endl;    // 输出非静态数据成员 score
}

void Student::show_count_sum_ave()    // 静态成员函数
{ cout<<"学生人数: "<<count<<endl;    // 输出静态数据成员 count
  cout<<"累加成绩: "<<sum<<endl;      // 输出静态数据成员 sum
  cout<<"平均成绩: "<<ave<<endl;      // 输出静态数据成员 ave
}

int Student::count=0;                 // 初始化静态数据成员 count
float Student::sum=0.0;                // 初始化静态数据成员 sum
float Student::ave=0.0;                // 初始化静态数据成员 ave

int main()
{ Student stu1("Liming", 90);
  Student::show(stu1);
  Student::show_count_sum_ave();       // 使用类名访问静态成员函数
  Student stu2("Zhanghao", 80);
  Student::show(stu2);
}
```

```

stu2.show_count_sum_ave();           // 使用对象名 stu2 访问静态成员函数
return 0;
}

```

本程序中例 4.9 中的非静态成员函数 show 改为静态成员函数 show，运行的结果与上例相同，如下：

```
姓名: Liming
```

```
成绩: 90
```

```
学生人数: 1
```

```
累加成绩: 90
```

```
平均成绩: 90
```

```
姓名: Zhanghao
```

```
成绩: 80
```

```
学生人数: 2
```

```
累加成绩: 170
```

```
平均成绩: 85
```

静态成员函数与非静态成员函数的重要的区别是：非静态成员函数有 this 指针，而静态成员函数没有 this 指针。静态成员函数可以直接访问本类中的静态数据成员，因为静态数据成员同样是属于类的，可以直接访问（如本例中的函数 show_count_sum_ave）。一般而言，静态成员函数不访问类中的非静态成员。假如在静态成员函数 show 中有以下语句：

```

cout<<"姓名: "<<name<<endl;           // 不合法, name 是非静态数据成员
cout<<"成绩: "<<score<<endl;           // 不合法, score 是非静态数据成员

```

若确实需要访问非静态数据成员，静态成员函数只能通过对象名（或对象指针、对象引用）访问该对象的非静态成员。如本例 show 函数定义为静态成员函数，这时可将对象的引用作为函数参数，将它定义为：

```

void Student::show(Student &stu)           // 静态成员函数
{ cout<<"姓名: "<<stu.name<<endl;           // 输出非静态数据成员 name
  cout<<"成绩: "<<stu.score<<endl;           // 输出非静态数据成员 score
}

```

下面对静态成员函数的使用再作几点说明。

(1) 一般情况下，静态成员函数主要用来访问静态数据成员。当它与静态数据成员一起使用时，达到了对同一个类中对象之间共享数据的目的。

(2) 私有静态成员函数不能被类外部的函数和对象访问。

(3) 使用静态成员函数的一个原因是，可以用它在建立对象之前调用静态成员函数，以处理静态数据成员，这是普通成员函数不能实现的功能。例如：

```

int main()
{
    Student::show_count_sum_ave(); // 可以用它在建立对象之前调用静态成员函数
    Student stu1("Liming", 90);
    ...
    return 0;
}

```

(4) 编译系统将静态成员函数限定为内部连接,也就是说,与现行文件相连接的其他文件中的同名函数不会与该函数发生冲突,维护了该函数使用的安全性,这是使用静态成员函数的另一个原因。

(5) 静态成员函数是类的一部分,而不是对象的一部分。如果要在类外调用公有的静态成员函数,使用如下格式较好:

类名::静态成员函数名()

如上例中的:

```
Student::show_count_sum_ave(); // 使用类名访问静态成员函数
```

当然,如果已经定义了这个类的对象(如 stu2),使用以下语句也是正确的:

```
stu2.show_count_sum_ave(); // 使用对象 stu2 访问静态成员函数
```

4.4 友元

类的主要特点之一是数据隐藏和封装,即类的私有成员只能在类定义的范围内使用,也就是说私有成员只能通过它的成员函数来访问。但是,有时为了访问类的私有成员而需要在程序中多次调用成员函数。带来的结果是:因为频繁调用带来较大的时间和空间开销,从而降低程序的运行效率。

为此,C++提供了一种访问私有成员的途径,在不放弃私有成员数据安全性的情况下,使得一个普通函数或者类的成员函数可以访问到封装于某一类中的信息(包括公有、私有、保护成员),在C++中用友元作为实现这个要求的辅助手段。C++中的友元为数据隐藏这堵不透明的墙开了一个小孔,外界可以通过这个小孔窥视类内部的秘密,友元是一扇通向私有成员的后门。

友元包括友元函数和友元类,下面分别予以介绍。

4.4.1 友元函数

友元函数既可以是属于任何类的非成员函数,也可以是另一个类的成员函数。友元函数不是当前类的成员函数,但它可以访问该类所有的成员,包括私有成员、保护成员和公有成员。

在类中声明友元函数时,需在其函数名前加上关键字 friend。此声明可以放在公有部分,也可以放在保护部分和私有部分。友元函数可以定义在类内部,也可以定义在类的外部。

1. 将非成员函数声明为友元函数

下面是一个将非成员函数声明为友元函数的例子。

例 4.13 非成员函数声明为友元函数。

```
#include<iostream>
using namespace std;
class Date{
public:
```

```

    Date(int y, int m, int d);           //声明构造函数
    friend void showDate(Date& d);       //声明函数 showDate 为类 Date 的友元函数
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d)          //定义构造函数, 给 year、month、day 赋初值
{
    year=y;
    month=m;
    day=d;
}
void showDate(Date& d)                   //定义友元函数 showDate, 形参是 Date 类对象的引用
{
    cout<<"日期: "<<d.year<<" "<<d.month<<" "<<d.day<<endl;
}
int main()
{
    Date date1(2010, 11, 14);
    showDate(date1);                     //调用友元函数 disp, 实参 e 是类 Girl 的对象
    return 0;
}

```

程序的运行结果如下:

日期: 2010, 11, 14

注意:

showDate 是一个在类外定义的非成员函数, 它是一个友元函数, 不属于任何类。它的作用是输出日期 (年、月、日)。从上面的例子可以看出, 友元函数可以访问类对象的各个私有数据。若在类 Date 的声明中将友元函数 showDate 的关键字 friend 去掉, 那么函数 showDate 对类对象的私有数据的访问将变为非法的。

说明:

(1) 友元函数虽然可以访问类对象的私有成员, 但它毕竟不是成员函数。因此, 在类的外部定义友元函数时, 不必像成员函数那样, 在函数名前加上“类名::”。

(2) 因为友元函数不是类的成员, 所以它不能直接访问对象的数据成员, 也不能通过 this 指针访问对象的数据成员, 它必须通过作为入口参数传递进来的对象名 (或对象指针、对象引用) 来访问该对象的数据成员。例如上面例子中的友元函数 showDate (Date& d) 的形参 d 是 Date 类的对象的引用, 此时函数体应写成

```
cout<<d.year<<" "<<d.month<<" "<<d.day<<endl;
```

(3) 由于函数 showDate 是 Date 类的友元函数, 所以 showDate 函数可以访问 Date 中的私有数据成员 year、month 和 day。但在访问 year、month 和 day 时, 必须加上对象名 d, 不能写成

```
cout<<year<<" "<<month<<" "<<day<<endl;
```

(4) 应该指出的是, 引入友元提高了程序运行效率, 实现了类之间的数据共享, 也方便编程。但是声明友元函数相当于在实现封装的黑盒子上开洞, 如果一个类声明了许多友元, 则相当于在黑盒子上开了很多洞, 显然这将破坏了数据的隐蔽性和类的封装性, 降低了程序的可维护性, 这与面向对象的程序设计思想是背道而驰的, 因此使用友元函数应谨慎。

2. 将成员函数声明为友元函数

除了非成员函数可以作为某个类的友元外，一个类的成员函数也可以作为另一个类的友元，它是友元函数中的一种，称为友元成员函数。友元成员函数不仅可以访问自己所在类对象中的私有成员和公有成员，还可以访问 `friend` 声明语句所在类对象中的所有成员，这样能使两个类相互合作、协调工作，完成某一任务。

在例 4.14 所列的程序中，声明了函数 `showDate_Time` 为类 `Time` 的成员函数，又是类 `Date` 的友元函数。

例 4.14 一个类的成员函数作为另一个类的友元函数。

```
#include<iostream>
using namespace std;
class Date;           //对 Date 类的提前引用声明
class Time{           //声明类 Boy
public:
    Time (int h, int m, int s)    //定义构造函数，给 hour、minute、sec 赋初值
    { hour =h;
      minute =m;
      sec =s;
    }
    void showDate_Time(Date&);    //声明函数 showDate_Time 为类 Time 的成员函数
private:
    int hour;
    int minute;
    int sec;
};

class Date{           //声明类 Date
public:
    Date(int y, int m, int d)    //定义构造函数，给 year、month、day 赋初值
    { year=y;
      month=m;
      day=d;
    }
    friend void Time::showDate_Time(Date&);    //声明类 Time 的成员函数
                                              //showDate_Time 为类 Date 的友元函数
private:
    int year;
    int month;
    int day;
};

void Time::showDate_Time (Date& d)    //定义类 Time 的成员函数 showDate_Time，同时
{                                     //也为类 Date 的友元函数，形参为 Date 类对象的引用
    cout<<"日期: "<<d.year<<"."<<d.month<<"."<<d.day<<endl; //作为 Date 类的
                                     //友元函数，函数 showDate_Time 可以访问 Date 类对象中的私有数据
    cout<<"时间: "<< hour <<":"<< minute <<":"<< sec <<endl; //作为 Time 类的
                                     //成员函数，函数 showDate_Time 可以访问 Time 类对象中的私有数据
}

int main()
{ Date date1(2010, 11, 14);    //定义 Date 类对象 date1
  Time time1(6, 12, 18);      //定义 Time 类对象 time1
```

```

    time1.showDate_Time (date1);    //调用 Time 类对象 time1 的成员函数和 Date
                                     //类的友元函数 showDate_Time, 实参是 Date 类对象 date1
    return 0;
}

```

程序的运行结果如下:

日期: 2010.11.14

时间: 6:12:18

说明:

(1) 一个类的成员函数作为另一个类的友元函数时, 必须先定义这个类。如例 4.14 中, 类 Time 的成员函数为类 Date 的友元函数, 必须先定义类 Time。在声明友元函数时, 要加上成员函数所在类的类名, 如:

```
friend void Time::showDate_Time(Date&);
```

(2) 程序中第 3 行语句 “class Date;” 为 Date 类的提前引用声明, 因为函数 showDate_Time 中将 “Date&” 作为参数, 而 Date 要在晚一些时候才被定义。

4.4.2 友元类

不仅可以将一个函数声明为一个类的友元函数, 而且可以将一个类 (例如 Y 类) 声明为另一个类 (例如 X 类) 的友元。这时 Y 类就是 X 类的友元类。友元类的说明方法是在另一个类声明中加入语句 “friend 类名”, 此语句可以放在公有部分, 也可以放在私有部分或保护部分。声明友元类的一般形式为:

friend 类名;

例如:

```

class Y {
    ...
};

class X {
    ...
    friend Y;    // 声明类 Y 为类 X 的友元类
    ...
};

```

当一个类被说明为另一个类的友元类时, 它的所有成员函数都成为另一个类的友元函数, 这就意味着作为友元的类中的所有成员函数都可以访问另一个类中的所有成员 (包括私有成员)。

下面的例子中, 声明了两个类 Date 和 Time, 类 Time 声明为类 Date 的友元, 因此类 Time 的成员函数都成为类 Date 的友元函数, 它们都可以访问类 Date 的私有成员。

例 4.15 一个类作为另一个类的友元类。

```

#include<iostream>
using namespace std;
class Date;    //对 Date 类的提前引用声明
class Time{    //声明类 Boy
public:
    Time (int h, int m, int s)    //定义构造函数, 给 hour、minute、sec 赋初值

```



```

    { hour =h;
      minute =m;
      sec =s;
    }

    void showDate_Time(Date&);           //声明函数 showDate_Time 为类 Time 的成员函数
private:
    int hour;
    int minute;
    int sec;
};

class Date{                             //声明类 Date
public:
    Date(int y, int m, int d)           //定义构造函数, 给 year、month、day 赋初值
    { year=y;
      month=m;
      day=d;
    }

    friend Time;                        //声明类 Time 为类 Date 的友元类
                                        //则类 Time 中的所有成员函数为类 Date 的友元函数

private:
    int year;
    int month;
    int day;
};

void Time::showDate_Time (Date& d)     //定义类 Time 的成员函数 showDate_Time
{                                       //同时也为类 Date 的友元函数, 形参为 Date 类对象的引用
    cout<<"日期: "<<d.year<<"."<<d.month<<"."<<d.day<<endl; //作为 Date 类的
                                        //友元函数, 函数 showDate_Time 可以访问 Date 类对象中的私有数据
    cout<<"时间: "<< hour <<" ":"<< minute <<" ":"<< sec <<endl; //作为 Time 类的
                                        //成员函数, 函数 showDate_Time 可以访问 Time 类对象中的私有数据
}

int main()
{ Date datel(2010, 11, 14);           //定义 Date 类对象 datel
  Time timel(6, 12, 18);              //定义 Time 类对象 timel
  timel.showDate_Time (datel);        //调用 Time 类对象 timel 的成员函数和 Date
                                        //类的友元函数 showDate_Time, 实参是 Date 类对象 datel

  return 0;
}

```

程序的运行结果如下:

日期: 2010.11.14

时间: 6:12:18

说明:

(1) 友元关系是单向的, 不具有交换性。若声明了类 X 是类 Y 的友元 (即在类 Y 定义中声明 X 为 friend 类), 不等于类 Y 一定是 X 的友元, 这要看类 X 中是否有相应的声明。

(2) 友元关系也不具有传递性, 若类 X 是类 Y 的友元, 类 Y 是类 Z 的友元, 不一定类 X 是类 Z 的友元。如果想让类 X 是类 Z 的友元类, 应在类 Z 中作出声明。

4.5 类的组合

前面已经讲过,复杂的对象可以由比较简单的对象以某种方式组合而成,复杂对象和组成它的简单对象之间的关系是组合关系。例如,计算机可构成计算机类,计算机类的数据成员有型号、CPU 参数、内存参数、硬盘参数、厂家等。其中的数据成员“厂家”又是计算机公司类的对象。这样,计算机类的数据成员中就有计算机公司类的对象,或者反过来说,计算机公司类的对象又是计算机类的一个数据成员。这样,当生成一个计算机类对象时,其中就嵌套着一个计算机公司对象。

在一个类中内嵌另一个类的对象作为数据成员,称为类的组合。该内嵌对象称为对象成员,也称为子对象。例如:

```
class Y
{ ...
};
class X
{
    Y y;    // 类 Y 的对象 y 为类 X 的对象成员
    ...
};
```

使用对象成员着重要注意的问题是,对象成员的初始化问题,即类 X 的构造函数如何定义的问题。当创建类的对象时,如果这个类具有内嵌的对象成员,那么内嵌对象成员也将被自动创建。因此,在创建对象时既要对本类的基本数据成员初始化,又要对内嵌的对象成员进行初始化。含有对象成员的类,其构造函数和不含对象成员的构造函数有所不同,例如有以下类 X:

```
class X{
    类名 1 对象成员名 1;
    ...
    类名 i 对象成员名 i;
    ...
    类名 n 对象成员名 n;
};
```

一般来说,类 X 的构造函数的定义形式为:

```
X::X(形参表 0):对象成员名 1(形参表 1), ..., 对象成员名 i(形参表 i), ..., 对象成员名 n(形参表 n)
{
    //类 X 的构造函数体
}
```

冒号后面的部分是对象成员的初始化列表,各对象成员的初始化列表用逗号分隔,形参表 i (i 为 1 到 n) 给出了初始化对象成员所需要的数据,它们一般来自形参表 0。

当调用构造函数 X()时,首先按各对象成员在类声明中的顺序依次调用它们的构造函数,

对这些对象初始化,最后再执行 X()的构造函数体初始化类中的其他成员。析构函数的调用顺序与构造函数的调用顺序相反。

下面我们看一个应用对象成员的例子。前面我们声明的学生类 Student 中,关于学生成绩只给出了一个数据成员 score,表示一门课的成绩。但实际上每个学生的学习成绩应含有多门课的成绩。所以,应该再多设置几个学习成绩的数据成员才更符合实际。考虑到所有学习成绩的性质和处理都是一致的,所以学习成绩也可单独作为一个类(成绩类),而把 Student 中的原成员 score 作为成绩类的一个对象。这样,一个学生类中就嵌套着一个成绩类对象。

例 4.16 学生类中嵌套着一个成绩类对象。

```
#include<iostream>
#include<string>
using namespace std;
class Score{                                //声明成绩类 Score
public:
    Score(double c=0, double e=0, double m=0);
    void show();
private:
    double computer;                        //计算机成绩
    double english;                        //英语成绩
    double mathematics;                    //数学成绩
};
Score::Score(double c, double e, double m)
{ computer = c;
  english = e;
  mathematics = m;
}
void Score::show()
{ cout<<"Score computer: "<<computer<<endl;
  cout<<"Score english: "<<english<<endl;
  cout<<"Score mathematics: "<<mathematics<<endl;
}
class Student{                             //声明学生类 Student
public:
    Student(string name1, double s1, double s2, double s3);
                                                //声明构造函数 Student
    void show();                             //声明输出数据函数 show
private:
    string name;                             //学生姓名
    Score score1;                            //类 Score 的对象 score1 是类 Student 的对象成员
};
Student::Student(string name1, double s1, double s2, double s3)
:score1(s1, s2, s3)                         //定义构造函数 Student, 缀上对象成员的初始化列表
{ name=name1;
}
void Student::show()                       //定义输出数据函数 show
{ cout<<"Name: "<<name<<endl;
  score1.show();
}
```

```
int main()
{ Student stu1("Liming", 85, 80, 70);
    // 定义类 Student 的对象 stu1, 调用 stu1 的构造函数, 初始化对象 stu1
    stu1.show(); // 调用 stu1 的 show 函数, 显示 stu1 的数据
    return 0;
}
```

本例构造函数的调用过程是:定义类 Student 的对象 stu1 时, 自动调用类 Score 的构造函数

```
Student::Student(string name1, double s1, double s2, double s3)
: score1(s1, s2, s3)
```

由该构造函数先自动通过 "score1 (s1, s2, s3)" 调用类 Score 的构造函数, 给对象成员的数据成员 computer、english 和 mathematics 赋值, 然后再执行类 Student 的构造函数体, 给数据成员 name 赋值。

程序的运行结果如下:

```
Name: Liming
Score computer: 85
Score english: 80
Score mathematics: 70
```

从上面的程序可以看出, 类 Student 的 show 函数中, 对于对象成员 score1 的处理就是通过调用类 Score 的 show 函数实现的。

说明:

(1) 声明一个含有对象成员的类, 首先要创建各成员对象。本例在声明类 Student 中, 定义了对象成员 score1:

```
Score score1;
```

(2) Student 类对象在调用构造函数进行初始化的同时, 也要对对象成员进行初始化, 因为它也是属于此类的成员。因此在写类 Student 的构造函数时, 也缀上了对对象成员的初始化:

```
Student::Student(string name1, double s1, double s2, double s3)
: score1(s1, s2, s3) { ... }
```

这时构造函数的调用顺序是: 先调用对象成员 score1 的构造函数, 随后再执行类 Student 构造函数的函数体。

这里需要注意的是: 在定义类 Student 的构造函数时, 必须缀上其对象成员的名字 score1, 而不能缀上类名, 若写成:

```
Student::Student(string name1, double s1, double s2, double s3)
: Score(s1, s2, s3) { ... }
```

是不允许的, 因为在类 Student 中是类 Score 的对象 score1 作为成员, 而不是类 Score 作为其成员。

4.6 共享数据的保护

虽然 C++ 采取了不少措施（如设置 `private` 数据等）来增加数据的安全性，但是有些数据是共享的，人们可以在不同场合通过不同的途径访问同一个数据对象。程序中各种形式的数据共享，在不同程度上破坏了数据的安全性。常类型的引入，就是为了既保证数据共享又防止数据被改动。常类型是指使用类型修饰符 `const` 说明的类型，常类型的变量或对象成员的值在程序运行期间是不可改变的。

4.6.1 常对象

如果在说明对象时用 `const` 修饰，则被说明的对象为常对象。常对象中的数据成员为常量且必须要有初值，如：

```
const Sample a(10, 20);           // a 是常对象，而不是普通对象
```

这样，常对象 `a` 中的数据成员值在对象的整个生存期内不能被改变。所谓对象的生存期是指对象从创建到被释放的时间间隔，也就是对象在程序中的作用域。常对象的说明形式如下：

类名 `const` 对象名[(参数表)];

或者

`const` 类名 对象名[(参数表)];

在定义对象时必须进行初始化，而且不能被更新。

例 4.17 非常对象和常对象的比较。

```
#include<iostream>
using namespace std;
class Date{
public:
    Date(int y, int m, int d);           //声明构造函数的原型
                                         //构造函数的名字必须与类名相同

    void setDate(int y, int m, int d);
    void showDate();
private:
    int year;
    int month;
    int day;
};
Date::Date(int y, int m, int d)         //定义构造函数 Date
{ year=y;
  month=m;
  day=d;
}
void Date::setDate(int y, int m, int d)
{ year=y;
  month=m;
  day=d;
}
```

```

}
void Date::showDate()
{ cout<<year<<". "<<month<<". "<<day<<endl; }
int main()
{ Date date1(2010, 4, 28);           //定义类 Date 的对象 date1, 自动调用构造函数
                                     //给对象 date1 的数据成员赋初值
    date1.showDate();                //调用成员函数 showDate, 显示 date1 的数据
    date1.setDate(2010, 12, 15);      //调用成员函数 setDate, 更改 date1 的数据成员
    date1.showDate();                //调用成员函数 showDate, 显示 date1 的数据
    return 0;
}

```

在这个例子中, 对象 `a` 是一个普通的对象, 而不是常对象, 读者不难分析程序的运行结果为:

```
2010.4.28
```

```
2010.12.15
```

若将上述程序中的对象 `date1` 定义为常对象, 主函数修改如下:

```

int main()
{ const Date date1(2010, 4, 28);      //语句①, 定义类 Date 的对象 date1, 构造函数
                                     //自动调用构造给对象 date1 的数据成员赋初值
    date1.showDate();                 //语句②, 显示 date1 的数据
    date1.setDate(2010, 12, 15);      //语句③, 更改 date1 的数据成员
    date1.showDate();                 //语句④, 显示 date1 的数据
    return 0;
}

```

编译这个程序时, 将出现 3 个错误。语句③的错误指出, C++不允许更改常对象的数据成员。语句②和④的错误指出, C++不允许常对象调用普通的成员函数。在 4.7.2 节中将会介绍, 常对象只能调用它的常成员函数。

4.6.2 常对象成员

C++可以在声明类时将其中的成员声明为 `const`, 即声明为常数据成员和常成员函数。

1. 常数据成员

类的数据成员可以是常量或常引用, 使用 `const` 说明的数据成员称为常数据成员。如果在一个类中说明了常数据成员, 那么构造函数就只能通过初始化列表对该数据成员进行初始化, 而任何其他函数都不能对该成员赋值。

例 4.18 常数据成员举例。

```

#include<iostream>
using namespace std;
class Date{
public:
    Date(int y, int m, int d);
    void showDate();
private:
    const int year;           //常数据成员
    const int month;          //常数据成员
}

```

```

        const int day;           //常数据成员
    };
    Date::Date(int y, int m, int d):year(y), month(m), day(d)
    { }                          //采用成员初始化列表,对常数据成员赋初值
    void Date::showDate()
    { cout<<year<<"."<<month<<"."<<day<<endl; }
    int main()
    { Date date1(2010, 10, 15);
      date1.showDate();
      return 0;
    }

```

程序运行结果如下:

```
2010.10.15
```

该程序中定义了如下3个常数据成员:

```

const int year
const int month;
const int day;

```

其中 year、month、day 是 int 类型的常数据成员。需要注意的是构造函数的格式如下:

```

Date::Date(int y, int m, int d):year(y), month(m), day(d)
{ }

```

其中,冒号后面是一个成员初始化列表,它包含3个初始化项。这是由于 year、month 和 day 都是常数据成员, C++ 规定只能通过构造函数的成员初始化列表对常数据成员进行初始化。在函数体中采用赋值语句对常数据成员直接赋初值是非法的,如以下形式的构造函数是错误的:

```

Date::Date(int y, int m, int d)
{ year=y;           //非法
  month=m;          //非法
  day=d;            //非法
}

```

注意:一旦对某对象的常数据成员初始化后,该数据成员的值是不能改变的,但不同对象中的该数据成员的值可以是不同的(在定义对象时给出)。

2. 常成员函数

在类中使用关键字 const 说明的函数为常成员函数,常成员函数的说明格式如下:

类型 函数名(参数表) const;

const 是函数类型的一个组成部分,因此在声明函数和定义函数时都要有关键字 const。但是在调用常成员函数时不必加关键字 const。

例 4.19 常成员函数的使用。

```

#include<iostream>
using namespace std;
class Date{
public:
    Date(int y, int m, int d);
    void showDate();           //声明普通成员函数 showDate
}

```

```

        void showDate() const;           //声明常成员函数 showDate
    private:
        int year;
        int month;
        int day;
};
Date::Date(int y, int m, int d):year(y), month(m), day(d)
{ }
void Date::showDate()                   //定义普通成员函数 showDate
{ cout<<"调用普通成员函数显示的日期:"<<endl;
  cout<<year<<". "<<month<<". "<<day<<endl;
}
void Date::showDate() const             //定义常成员函数 showDate
{ cout<<"调用常成员函数显示的日期:"<<endl;
  cout<<year<<". "<<month<<". "<<day<<endl;
}
int main()
{ Date date1(2010, 4, 28);              //定义普通对象 date1
  date1.showDate();                     //调用普通成员函数 showDate
  const Date date2(2010, 11, 14);       //定义常对象 date2
  date2.showDate();                     //调用常成员函数 showDate
  return 0;
}

```

程序运行结果如下:

调用普通成员函数显示的日期:

2010.4.28

调用常成员函数显示的日期:

2010.11.14

本程序中, 类 `Date` 中说明了两个同名成员函数 `showDate`, 一个是普通的成员函数, 另一个是常成员函数, 它们是重载的。可见, 关键字 `const` 可以被用于对重载函数的区分。在主函数中说明了两个对象 `date1` 和 `date2`, 其中对象 `date2` 是常对象。通过对象 `date1` 调用的是没有用 `const` 修饰的普通成员函数, 而通过常对象 `date2` 调用的是用 `const` 修饰的常成员函数。

4.7 C++的多文件程序

我们已经学习到了很多完整的 C++ 源程序实例, 分析它们的结构, 基本上都是由 3 部分构成: 类的声明部分、类的实现部分和类的使用部分。因为前面我们所举的例子都比较小, 所以这 3 部分都写在同一个文件中。

在实际程序设计中, 一个源程序按照结构可以划分为 3 个文件: 类声明文件 (*.h 文件)、类实现文件 (*.cpp) 和类的使用文件 (*.cpp, 主函数文件)。将类的声明部分放在类声明文件(头文件)中, 类中向用户提供成员函数所需的函数原型, 这就形成了类的 `public` 外部接口。将类成员函数的定义放在类实现文件中, 这就形成了类的实现方法。将类的使用部分(通常是主程序)放在类使用文件中, 这样可以清晰地表示出本程序所要完成的工作。下面的例

4.20 将按照这样的原则，划分为3个文件：student.h（类声明文件）、student.cpp（类实现文件）和 studentmain.cpp（类使用文件）。完整的程序如下例所示。

例 4.20 一个源程序按照结构划分为3个文件。

```
// 文件1 student.h（类的声明部分）
#include<iostream>
using namespace std;
class Student{
public:
    Student(char *name1, char *stu_no1, float score1);           //类的外部接口
    ~Student();           //声明构造函数
    void modify(float score1);           //声明析构函数
    void disp();           //声明数据修改函数
private:
    char *name;           //声明数据输出函数
    char *stu_no;
    float score;
};
// 文件2 student.cpp（类的实现部分）
#include "student.h"           //包含类的声明文件
Student::Student(char *name1, char *stu_no1, float score1)       //构造函数的实现
{ name=new char[strlen(name1)+1];
  strcpy(name, name1);
  stu_no=new char[strlen(stu_no1)+1];
  strcpy(stu_no, stu_no1);
  score=score1;
}
Student::~~Student()           //析构函数的实现
{ delete []name;
  delete []stu_no;
}
void Student::modify(float score1)           //数据修改函数的实现
{ score=score1; }
void Student::disp()           //数据输出函数的实现
{ cout<<"name: "<<name<<endl;
  cout<<"stu_no: "<<stu_no<<endl;
  cout<<"score: "<<score<<endl;
}
// 文件3 studentmain.cpp（类的使用部分）
#include "student.h"           //包含类的声明文件
int main()
{ Student stu1("Liming", "20080201", 90);           //定义类 Student 的对象 stu1
    //调用 stu1 的构造函数，初始化对象 stu1
    stu1.disp();           //调用 stu1 的成员函数 disp，显示 stu1 的数据
    stu1.modify(88);           //调用 stu1 的成员函数 modify，修改 stu1 的数据
    stu1.disp();           //调用 stu1 的成员函数 disp，显示 stu1 修改后的数据
    return 0;
}
```

程序运行结果如下：

```
name: Liming
stu_no: 20080201
score: 90
name: Liming
stu_no: 20080201
score: 88
```

由于类的声明和实现放在两个不同的文件 `student.h` 和 `student.cpp` 中，在类的实现文件中就必须包含类的声明文件 `student.h`。把类的声明和实现放在不同的文件之中，主要有以下考虑。

(1) 类的实现文件通常较大，将两者混在一起不便于阅读、管理和维护。一个良好的软件工程基本原则是将接口与实现方法分离，这样可以更容易地修改程序。对类的用户而言，类的实现方法的改变并不影响用户，只要接口不变即可。

(2) 将类中成员函数的实现放在其声明文件（如 `student.h`）中与放在实现文件（如 `student.cpp`）中，在编译时的含义是不一样的，若将成员函数的实现直接放在类的声明中，则类的成员函数将作为内联函数处理。显然将所有的成员函数都作为内联函数处理是不合适的。

(3) 对于软件开发商来说，他们可以向用户提供一些程序模块，这些程序模块往往只向用户公开类的声明（即接口），而不公开程序的源代码。而类的用户使用类时不需要访问类的源代码，但需要连接类的目标码。类的声明和实现分开管理可以很好地解决这个问题。

(4) 便于团体式的大型软件开发。采用这样的组织结构，可以对各个文件进行单独编辑、编译，最后再连接和运行。同时可以充分利用类的封装特性，在程序的调试、修改时只对其某一个文件进行操作，而其余部分根本就不用改动。例如，我们只修改了类的成员函数的实现部分，则只需重新编译类实现文件并连接即可，其余的文件几乎可以不看。如果是一个语句很多、规模特大的程序，效率就会得到显著的提高。

由多个文件组成的程序的编辑、编译、连接和执行方法请参阅附录。

在此需要说明的是，由于本书大部分程序较小，为了节省篇幅，就不分成 3 部分编写了，对部分较大的程序将采用这种结构编写。

4.8 应用举例

例 4.21 利用类表示一个堆栈（stack），并为此堆栈建立向堆栈压入数据函数 `push`、从堆栈中弹出数据函数 `pop` 及显示堆栈数据函数 `showstack` 等函数。

整个程序分为 3 个独立文件：`Stack.h` 是类声明头文件，`Stack.cpp` 是类实现文件，`Stackmain.cpp` 是类的使用文件。

```
// 类声明头文件 Stack.h
#include<iostream>
using namespace std;
const int SIZE=10;
```

```

class Stack{
public:
    Stack();
    void push(int ch);           //声明向堆栈压入数据函数
    int pop();                   //声明从堆栈中弹出数据函数
    void ShowStack();           //声明显示堆栈数据函数
    int interface();
private:
    int stck[SIZE];             // 数组, 用于存放栈中数据
    int tos;                     // 栈顶位置 (数组下标)
};

//类实现文件 Stack.cpp
#include "Stack.h"
Stack::Stack()                 // 构造函数, 初始化栈的实现
{ tos= 0; }
void Stack::push(int ch)       //向堆栈压入数据函数的实现
{ if(tos==SIZE)
    { cout<<"Stack is full";
      return;
    }
    stck[tos]=ch;
    tos++;
    cout<<"You have pushed a data into the stack!\n";
}
int Stack::pop()               //从堆栈中弹出数据函数的实现
{ if (tos==0)
    { cout<<"Stack is empty";
      return 0;
    }
    tos--;
    return stck[tos];
}
void Stack::ShowStack()        //显示堆栈数据函数
{ cout<<"The content of stack: \n" ;
  if (tos==0)
  { cout<<"The stack has no data!\n";
    return;
  }
  for (int i=tos-1; i>=0;i--)
    cout<<stck[i]<<" ";
    cout<<"\n";
}
int Stack::interface()
{
    int x;
    char ch;
    cout<<" <I> ----- Push data to stack\n";
    cout<<" <O> ----- Pop data from stack\n";
    cout<<" <S> ----- Show the content of stack\n";
    cout<<" <Q> ----- Quit... \n";
    while (1)
    { cout<<"Please select an item: ";
      cin>>ch;

```

```

    ch=toupper(ch);
    switch(ch)
    { case 'I':
      cout<<"Enter the value that "<<"you want to push: ";
      cin >>x;
      push(x);
      break;
    case 'O':
      x=pop();
      cout<<"Pop "<<x<<" from stack.\n";
      break;
    case 'S':
      ShowStack();
      break;
    case 'Q':
      return 0;
    default:
      cout<<"You have inputted a wrong item! Please try again!\n";
      continue;
    }
  }
  return 0;
}
//类的使用文件 Stackmain.cpp
#include "Stack.h"
int main()
{ cout<<endl;
  Stack ss;
  ss.interface();
  return 0;
}

```

本程序的一次运行结果如下:

```

<I> ----- Push data to stack
<O> ----- Pop data from stack
<S> ----- Show the content of stack
<Q> ----- Quit...

Please select an item: I
Enter the value that you want to push: 10
You have pushed a data into the stack!

Please select an item: D
You have inputted a wrong item! Please try again!

Please select an item: I
Enter the value that you want to push: 20
You have pushed a data into the stack!

Please select an item: I
Enter the value that you want to push: 30
You have pushed a data into the stack!

```

Please select an item: S

The content of stack:

30 20 10

Please select an item: O

Pop 30 from stack.

Please select an item: O

Pop 20 from stack.

Please select an item: O

Pop 10 from stack.

Please select an item: S

The content of stack:

The stack has no data!

Please select an item: Q

说明:

在本程序中声明了一个堆栈类 Stack, 类中定义了数据成员 tos 和数组 stck。类中 SIZE 表示堆栈的最大存储空间; tos 表示栈顶; 数组 stck 用来存放堆栈数据。类中还定义了成员函数 push 用来向堆栈压入数据, pop 用来从堆栈中弹出数据, ShowStack 从栈顶到栈底显示堆栈的内容。在主程序 main 中通过按不同的键, 调用不同的成员函数, 以实现不同的功能。

实 验

实验目的和要求

1. 学会使用 Visual C++ 6.0 编辑、编译、连接和运行 C++ 的多文件程序的方法。
2. 熟悉对象数组和对象指针的使用方法。
3. 学习使用对象、对象指针和对象引用作为函数参数的方法。
4. 学习类对象作为成员的使用方法。
5. 学习静态数据成员和静态成员函数的使用方法。
6. 理解友元的概念和学习友元的使用方法。

实验内容和步骤

1. 编辑、编译、连接和运行以下的 C++ 多文件程序。

```
// file1.cpp
#include<iostream>
using namespace std;
int add(int a, int b);
int main()
{ int x, y, sum;
  cout<<"Enter two numbers:"<<endl;
  cin>>x;
  cin>>y;
  sum=add(x, y);
  cout<<"The sum is: "<<sum<<endl;
```

```

    return 0;
}
//file2.cpp
int add(int a, int b)
{ int c;
  c=a+b;
  return c;
}

```

2. 调试下列程序，写出输出结果。

```

//test4_2.cpp
#include<iostream>
using namespace std;
class toy
{ public:
    toy(int q, int p)
    { quan = q;
      price = p;
    }
    int get_quan()
    { return quan;}
    int get_price()
    { return price;}
private:
    int quan, price;
};
int main()
{ toy op[3][2]={
    toy(10, 20), toy(30, 48),
    toy(50, 68), toy(70, 80),
    toy(90, 16), toy(11, 120),
};
for (int i=0;i<3;i++)
{ cout<<op[i][0].get_quan()<<" ";
  cout<<op[i][0].get_price()<<"\n";
  cout<<op[i][1].get_quan()<<" ";
  cout<<op[i][1].get_price()<<"\n";
}
cout<<endl;
return 0;
}

```

3. 设计一个用来表示直角坐标系的 Location 类，在主程序中创建类 Location 的两个对象 A 和 B，要求 A 的坐标点在第 3 象限，B 的坐标点在第 2 象限，分别采用成员函数和友元函数计算给定两个坐标点之间的距离，要求按如下格式输出结果：

```

A(x1, y1), B(x2, y2)
Distance1=d1
Distance2=d2

```

其中，x1、y1、x2、y2 为指定的坐标值，d1 和 d2 为两个坐标点之间的距离。

【提示】

类 Location 的参考框架如下:

```
class Location {
public:
    Location(double, double);           //构造函数
    double Getx();                       //成员函数, 取 x 坐标的值
    double Gety();                       //成员函数, 取 y 坐标的值
    double distance(Location&);          //成员函数, 求给定两点之间的距离
    friend double distance(Location &, Location &); //友元函数, 求给定两点之间的距离

private:
    double x, y;
};
```

4. 声明一个 Student 类, 在该类中包括一个数据成员 score (分数)、两个静态数据成员 total_score (总分) 和 count (学生人数); 还包括一个成员函数 account 用于设置分数、累计学生的成绩之和、累计学生人数, 一个静态成员函数 sum 用于返回学生的成绩之和, 另一个静态成员函数 average 用于求全班成绩的平均值。在 main 函数中, 输入某班同学的成绩, 并调用上述函数求出全班学生的成绩之和与平均分。

习 题

【4.1】什么是对象数组?

【4.2】什么是对象指针? 声明对象指针的一般语法形式是什么?

【4.3】假定有一个类, 类名为 Date, 则执行 “Date d1(20), d2[5];” 语句时, 自动调用该类构造函数的次数为 ()。

- A. 25 B. 6 C. 2 D. 21

【4.4】下面对静态数据成员的描述中, 正确的是 ()。

- A. 类的不同对象有不同的静态数据成员值
B. 类的每个对象都有自己的静态数据成员
C. 静态数据成员是类的所有对象共享的数据
D. 静态数据成员不能通过类的对象调用

【4.5】在下面有关静态成员函数的描述中, 正确的是 ()。

- A. 在静态成员函数中可以使用 this 指针
B. 在建立对象前, 就可以为静态数据成员赋值
C. 静态成员函数在类外定义时, 要用 static 前缀
D. 静态成员函数只能在类外定义

【4.6】下列选项中, 静态成员函数不能直接访问的是 ()。

- A. 静态数据成员 B. 静态成员函数
C. 类以外的函数和数据 D. 非静态数据成员

【4.7】下列选项中, () 不是类的成员函数。

- A. 构造函数 B. 析构函数
C. 友元函数 D. 拷贝构造函数

【4.8】一个类的友元函数或友元类能够访问该类的()。

- A. 公用成员 B. 保护成员
C. 私有成员 D. 公用成员、保护成员和私有成员

【4.9】下面关于友元的描述中，错误的是（ ）。

- A. 类与类之间的友元关系可以继承
B. 一个类的友元类中的成员函数都是这个类的友元函数
C. 友元可以提高程序的运行效率
D. 友元函数可以访问该类的私有数据成员

【4.10】友元的作用之一是()。

- A. 提高程序的运行效率 B. 加强类的封装性
C. 实现数据的隐藏性 D. 增加成员函数的种类

【4.11】对于常成员函数，下面描述正确的是（ ）。

- A. 常成员函数不能修改任何数据成员 B. 常成员函数只能修改一般数据成员
C. 常成员函数只能修改常数据成员 D. 常成员函数只能修改变量的数据成员

【4.12】以下程序的运行结果是()。

```
#include<iostream>
using namespace std;
class B{
public:
    B() { }
    B(int i, int j)
    { x=i; y=j; }
    void printb()
    { cout<<x<<" ", <<y<<endl; }
private:
    int x, y;
};
class A{
public:
    A() { }
    A(int i, int j);
    void printa();
private:
    B c;
};
A::A(int i, int j):c(i, j)
{ }
void A::printa()
{ c.printb();}
int main()
{ A a(7, 8);
  a.printa();
  return 0;
}
```

- A. 8, 9 B. 7, 8 C. 5, 6 D. 9, 10

【4.13】以下程序的运行结果是()。

```
#include<iostream>
using namespace std;
class Sample{
public:
    Sample( int i, int j)
    { x=i; y=j; }
    void disp()
    { cout<<"disp1"<<endl; }
    void disp() const
    { cout<<"disp2"<<endl; }
private:
    int x, y;
};
int main()
{ const Sample a(1, 2);
  a.disp();
  return 0;
}
```

A. disp1 B. disp2 C. disp1 disp2 D. 程序编译出错

【4.14】指出下面程序中的错误，并说明原因。

```
#include<iostream>
using namespace std;
class CTest{
public:
    CTest()
    { x=20; }
    void use_friend();
private:
    int x;
    friend void friend_f(CTest fri);
};
void friend_f(CTest fri)
{ fri.x=55; }
void CTest::use_friend()
{ CTest fri;
  this->friend_f(fri);
  ::friend_f(fri);
}
int main()
{ CTest fri, fri1;
  fri.friend_f(fri);
  friend_f(fri1);
  return 0;
}
```

【4.15】指出下面程序中的错误，并说明原因。

```
#include<iostream>
using namespace std;
class CTest{
public:
    const int y2;
```

```

    CTest (int i1, int i2):y1(i1), y2(i2)
    { y1=10;
      x=y1;
    }
    int readme() const;
        //...
private:
    int x;
    const int y1;
};
int CTest::readme() const
{ int i;
  i=x;
  x++;
  return x;
}
int main()
{ CTest c(2, 8);
  int i=c.y2;
  c.y2=i;
  i=c.y1;
  return 0;
}

```

【4.16】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
class Test{
public:
    Test();           //不带参数的构造函数
    Test(int n);      //带1个参数构造函数
private:
    int num;
};
Test::Test()
{ cout<<"Init defa"<<endl;
  num=0;
}
Test::Test(int n)
{ cout<<"Tnit"<<" "<<n<<endl;
  num=n;
}
int main()
{ Test x[3];
  Test y(35);
  return 0;
}

```

【4.17】建立一个对象数组，内放 6 个学生的数据（学号、成绩），用指针指向数组首元素，输出第 2，4，6 个学生的数据。

【4.18】建立一个对象数组，内放 6 个学生的数据（学号、成绩），设立一个函数 max，使用对象指针作为函数参数，在 max 函数中找出 6 个学生中成绩最高者，并输出其学号。

【4.19】构建一个类 book，其中含有两个私有数据成员 qu 和 price，建立一个有 5 个元素

的数组对象,将 `qu` 初始化为 1~5,将 `price` 初始化为 `qu` 的 10 倍。显示每个对象的 `qu*price`。

【4.20】修改题 4.19,通过对象指针访问对象数组,使程序以相反的顺序显示对象数组的 `qu*price`。

【4.21】编写一个程序,已有若干学生的数据,包括学号、姓名、成绩,要求输出这些学生的数据并计算出学生人数和平均成绩(要求将学生人数和总成绩用静态数据成员表示)。

【4.22】设计一个点类,其中包含一对坐标点数据成员、一个求两个点之间距离的友元函数 `dist` 和显示坐标点的成员函数,并用数据进行测试。

第 5 章

继承与派生

继承是面向对象程序设计的一个重要特性，是软件复用的一种形式，它允许在已有类的基础上创建新的类。新类可以从一个或多个已有类中继承数据和函数，并且可以重新定义或增加新的数据和函数，从而引成类的层次或等级。其中已有类称为基类或父类，在它基础上建立的新类称为派生类或子类。可以说，如果没有掌握继承，就等于没有掌握类和对象的精华，就是没有掌握面向对象程序设计的真谛。

5.1 继承与派生的基本概念

5.1.1 为什么要使用继承

继承性是一个非常自然的概念，现实世界中的许多事物都具有继承性。人们一般用层次分类的方法来描述它们的关系。例如，图 5.1 所示为一个简单的学生分类图。



图 5.1 简单的学生分类图

在这个分类图中建立了一个层次结构，最高层是最普遍、最一般的，每一层都比它的前一层更具体，低层含有高层的特性，同时也与高层有细微的不同，它们之间是基类和派生类的关系。

继承就是从先辈处得到属性和行为特征。类的继承就是新的类从已有类那里得到已有的特性。从另一个角度来看这个问题，从已有类产生新类的过程就是类的派生。类的继承和派生机制使程序员无需修改已有类，只需在已有类的基础上，通过增加少量代码或修改少量代码的方法得到新的类，从而较好地解决了代码重用的问题。由已有类产生新类时，新类便包含了已有类的特征，同时也可以加入自己的新特性。已有类称为基类或父类，产生的新类称为派生类或子类。派生类同样也可以作为基类派生出新的类，这样就形成了类的层次结构。

关于基类和派生类的关系，可以表述为：派生类是基类的具体化，而基类则是派生类的抽象。从图 5.1 可以看到小学、中学、大学和研究生是学生的具体化，他们是在学生的共性基础上加上某些特点形成的子类。而学生类则是各类学生共性的综合，是对各类具体学生特点的抽象。基类综合了派生类的公共特征，派生类则在基类的基础上增加某些特性，把抽象类变成具体、实用的类型。

下面我们通过例子进一步说明为什么要使用继承。现有一个学生类 `Student`，它含有 `number`(学号)、`name`(姓名)、`score`(成绩)等数据成员与成员函数 `print`，如下所示：

```
class Student{                                //声明学生 Student
public:
    ...
    void print()
    { cout<<"number:"<<number<<endl;
      cout<<"name:"<<name<<endl;
      cout<<"score:"<<score<<endl;
    }
protected:
    int number;                               //学号
    string name;                             //姓名
    float score;                             //成绩
};
```

假如现在要声明一个大学生类 `UStudent`，它含有 `number` (学号)、`name` (姓名)、`score` (成绩) 及 `major` (专业) 等数据成员与成员函数 `print1`，如下所示：

```
class UStudent{                              //声明大学生类 UStudent
public:
    ...
    void print1()
    { cout<<"number:"<<number<<endl;        //此行在类 Student 中已存在
      cout<<"name:"<<name<<endl;            //此行在类 Student 中已存在
      cout<<"score:"<<score<<endl;          //此行在类 Student 中已存在
      cout<<"major:"<<major<<endl;
    }
private:
    int number;                               //学号，此行在类 Student 中已存在
    string name;                             //姓名，此行在类 Student 中已存在
    float score;                             //成绩，此行在类 Student 中已存在
    string major;                             //专业，此行在类 Student 中已存在
};
```

从以上两个类的声明中看出，这两个类中的数据成员和成员函数有许多相同的地方。只要在 `Student` 类的基础上再增加数据成员 `major`，再对 `print1` 成员函数稍加修改就可以声明出 `UStudent` 类。像现在这样声明两个类，代码重复太严重。为了提高代码的可重用性，可以引入继承，将 `UStudent` 类说明成 `Student` 类的派生类，那些相同的成员在 `UStudent` 类中就不需要再说明了。

说明：

在类 `Student` 中，我们使用了关键字 `protected`，将相关的数据成员说明成保护成员。保

护成员可以被本类的成员函数访问，也可以被本类的派生类的成员函数访问，而类以外的任何访问都是非法的，即它是半隐藏的。关于保护成员的特性将在后续章节中详细介绍。

5.1.2 派生类的声明

为了理解一个类如何继承另一个类，我们看一下 UStudent 类是如何继承 Student 类的。

```
class Student{                                //声明基类 Student
public:
    ...
    void print()
    { cout<<"number:"<<number<<endl;
      cout<<"name:"<<name<<endl;
      cout<<"score:"<<score<<endl;
    }
protected:
    int number;                               //学号
    string name;                             //姓名
    float score;                             //成绩
};

class UStudent:public Student{               //声明派生类 UStudent 公有继承了基类 Student
public:
    ...
    void print1()                            //新增加的成员函数
    { print();
      cout<<"major:"<<major<<endl;
    }
private:
    string major;                            //专业, 新增加的数据成员
};
```

仔细分析以上两个类，不难发现，在“class UStudent:”之后，跟着关键字 public 与类名 Student，这就意味着类 UStudent 继承了类 Student。其中类 Student 是基类，类 UStudent 是派生类。关键字 public 指出基类 Student 中的成员在派生类 UStudent 中的继承方式。基类名前面有 public 的继承称为公有继承。

声明一个派生类的一般格式为：

```
class 派生类名:[继承方式] 基类名 {
    派生类新增的数据成员和成员函数
};
```

这里，“基类名”是一个已经声明的类的名称，“派生类名”是继承原有类的特性而生成的新类的名称。“继承方式”规定了如何访问从基类继承的成员，它可以是关键字 private、protected 或 public，分别表示私有继承、保护继承和公有继承。因此，由类 Student 继承出类 UStudent 可以采用下面的 3 种格式之一：

(1) 公有继承

```
class UStudent:public Student{
    ...
};
```

(2) 私有继承

```
class UStudent:private Student{
    ...
};
```

(3) 保护继承

```
class UStudent:protected Student{
    ...
};
```

如果不显式地给出继承方式关键字,系统默认为私有继承(private)。类的继承方式指定了派生类成员以及类外对象对于从基类继承来的成员的访问权限。

派生类除了可以从基类继承成员外,还可以增加自己的数据成员和成员函数。这些新增的成员正是派生类不同于基类的关键所在,是派生类对基类的发展。

从已有类派生出新类时,可以在派生类内完成以下几种功能:

- (1) 可以增加新的数据成员;
- (2) 可以增加新的成员函数;
- (3) 可以对基类的成员进行重定义;
- (4) 可以改变基类成员在派生类中的访问属性。

这些内容将在后续章节详细介绍。

5.1.3 基类成员在派生类中的访问属性

派生类可以继承基类中除了构造函数与析构函数之外的成员,但是这些成员的访问属性在派生过程中是可以调整的。从基类继承来的成员在派生类中的访问属性是由继承方式控制的。

类的继承方式有 public (公有继承)、protected (保护继承) 和 private (私有继承) 3 种,不同的继承方式导致不同访问属性的基类成员在派生类中的访问属性也有所不同。

在派生类中,从基类继承来的成员可以按访问属性划分为 4 种,即不可直接访问、公有(public)、保护(protected)和私有(private)。表 5.1 列出了基类成员在派生类中的访问属性。

表 5.1 基类成员在派生类中的访问属性

基类中的成员	继承方式	基类成员在派生类中的访问属性
私有成员(private)	公有继承(public)	不可直接访问
私有成员(private)	私有继承(private)	不可直接访问
私有成员(private)	保护继承(protected)	不可直接访问
公有成员(public)	公有继承(public)	公有(public)
公有成员(public)	私有继承(private)	私有(private)
公有成员(public)	保护继承(protected)	保护(protected)
保护成员(protected)	公有继承(public)	保护(protected)
保护成员(protected)	私有继承(private)	私有(private)
保护成员(protected)	保护继承(protected)	保护(protected)

从表 5.1 中不难归纳出以下几点。

(1) 基类中的私有成员。

无论哪种继承方式，基类中的私有成员都不允许派生类继承，即在派生类中是不可直接访问的。

(2) 基类中的公有成员。

- 公有继承时，基类中的公有成员在派生类中仍以公有成员的身份出现。
- 私有继承时，基类中的公有成员在派生类中都是以私有成员的身份出现的。
- 保护继承时，基类中的公有成员在派生类中都是以保护成员的身份出现的。

(3) 基类中的保护成员。

- 公有继承时，基类中的保护成员在派生类中仍以保护成员的身份出现。
- 私有继承时，基类中的保护成员在派生类中都是以私有成员的身份出现的。
- 保护继承时，基类中的保护成员在派生类中仍以保护成员的身份出现。

5.1.4 派生类对基类成员的访问规则

基类的成员可以有 public（公有）、protected（保护）和 private（私有）3 种访问属性，基类的成员函数可以访问基类中其他成员，但是在类外通过基类的对象，就只能访问该基类的公有成员。同样，派生类的成员也可以有 public（公有）等 3 种访问属性，派生类的成员函数可以访问派生类中自己增加的成员，但是在派生类外通过派生类的对象，就只能访问该派生类的公有成员。

通过 5.1.3 节分析，我们知道类的继承方式有 public（公有继承）、protected（保护继承）和 private（私有继承）3 种，不同的继承方式导致原来具有不同访问属性的基类成员在派生类中的访问属性也有所不同。本节将介绍派生类对基类成员的访问规则。派生类对基类成员的访问形式主要有以下两种。

- (1) 内部访问：由派生类中新增的成员函数对基类继承来的成员的访问；
- (2) 对象访问：在派生类外部，通过派生类的对象对从基类继承来的成员的访问。

下面具体讨论在 3 种继承方式下，派生类对基类成员的访问规则。

1. 私有继承的访问规则

通过表 5.1 可以看出，当类的继承方式为私有继承时，基类的公有成员和保护成员被继承后作为派生类的私有成员，派生类的成员函数可以直接访问它们，但是在类外部通过派生类的对象无法访问。基类的私有成员不允许派生类继承，因此在私有派生类中是不可直接访问的，所以无论是派生类成员函数还是通过派生类的对象，都无法直接访问从基类继承来的私有成员。表 5.2 总结了私有继承的访问规则。

表 5.2 私有继承的访问规则

基类中的成员		私有成员	公有成员	保护成员
访问方式	内部访问	不可访问	可访问	可访问
	对象访问	不可访问	不可访问	不可访问

下面是一个私有继承的例子。

例 5.1 私有继承的访问规则示例 1。

```
#include <iostream>
using namespace std;
```



```

class B{                                //声明基类 B
public:
    void getx(int n)                    //正确, 基类的成员函数可以访问本类的私有成员 x
    { x=n; }
    void dispX()                        //正确, 基类的成员函数可以访问本类的私有成员 x
    { cout<<x<<endl; }
private:
    int x;
};
class D:private B{                      //声明基类 B 的私有派生类 D
public:
    void getxy(int n, int m)
    { getx(n);                          //基类的 getx 函数在派生类中为私有成员, 派生类成员函数可以访问
      y=m;                              //正确, 派生类的成员函数可以访问本类的私有成员 y
    }
    void dispxy()
    { cout<<x<<endl;                    //错误, 派生类成员函数不能直接访问基类的私有成员 x
      cout<<y<<endl;                    //正确, 派生类的成员函数可以访问本类的私有成员 y
    }
private:
    int y;
};
int main()
{ D obj;
  obj.getx(11);                         //错误, 函数 getx 在派生类中为私有成员, 派生类对象不能访问
  obj.dispx();                          //错误, 函数 dispx 在派生类中为私有成员, 派生类对象不能访问
  obj.getxy(22, 33);                    //正确, 函数 getxy 在类 D 为公有成员, 派生类对象能访问
  obj.dispxy();                          //正确, 函数 dispxy 在类 D 为公有成员, 派生类对象能访问
  return 0;
}

```

本例中首先定义了一个类 B, 它有一个私有数据成员 x 和两个公有成员函数 getx 和 dispx。将类 B 作为基类, 派生出一个类 D。派生类 D 除继承了基类的成员外, 还有只属于自己的成员, 即私有数据成员 y, 公有函数成员 getxy 和 dispxy。继承方式关键字是 private, 所以这是一个私有继承。

由于是私有继承, 所以基类 B 的公有成员函数 getx 和 dispx 被派生类 D 私有继承后, 成为派生类 D 的私有成员, 只能被 D 的成员函数访问, 不能被派生类的对象访问。所以在 main 函数中, 对 obj.getx() 和 obj.dispx() 的调用是非法的, 因为这两个函数在派生类 D 中已成为私有成员。

需要注意的是: 无论函数 getx 和 dispx 如何被一些派生类继承, 它们仍然是 B 的公有成员, 因此在 main 函数中以下的调用是合法的:

```

B base_obj;
base_obj.getx(2);

```

虽然派生类 D 私有继承了基类 B, 但它的成员函数并不能直接访问 B 的私有数据 x, 只能访问两个公有成员函数。所以在类 D 的成员函数 getxy 中访问 B 的公有成员函数 getx 是合法的, 但在成员函数 dispxy 中直接访问 B 的私有成员 x 是非法的。

如果将例中函数 `dispxy()` 改成如下形式:

```
void dispxy()
{ dispx();
  cout<<y<<endl;
}
```

重新编译, 程序将顺利通过。可见基类中的私有成员既不能被派生类的对象访问, 也不能被派生类的成员函数访问, 只能被基类自己的成员函数访问。因此, 我们在设计基类时, 总要为它的私有数据成员提供公有成员函数, 如本例的成员函数 `dispx` 等, 以便使派生类可以间接访问这些数据成员。

修改后, 程序运行结果如下:

```
22 例 5.2 私有继承的访问规则示例 2
33
```

下面程序说明基类中的保护成员以私有方式被继承后的访问属性。

例 5.2 私有继承的访问规则示例 2。

```
#include<iostream>
using namespace std;

class B{                                // 声明一个基类 B
public:
    void getB(int sa)                  // 正确, 基类的成员函数可以访问本类的私有成员 a
    { a=sa; }
    void dispB()                       // 正确, 基类的成员函数可以访问本类的私有成员 a
    { cout<<"a="<<a<<endl; }
protected:
    int a;
};

class D1: private B{                  // 声明一个私有派生类 D1
public:
    void getD1(int sa)                 // 正确, 基类的保护成员 a 在派生类中为私有成员
    { a=sa; }                          // 可以被派生类的成员函数访问
    void dispD1()                     // 正确, 基类的保护成员 a 在派生类中为私有成员
    { cout<<"a="<<a<<endl; }          // 可以被派生类的成员函数访问
};

class D2: private D1{                 // 声明一个私有派生类 D2
public:
    void getD2(int sa)                 // 正确, 派生类 D1 的函数 getD1 在派生类 D2 中
    { getD1(sa); }                    // 为私有成员, 可以被派生类 D2 的成员函数访问
    void dispD2()                     // 错误, a 在派生类 D2 中为不可直接访问成员
    { cout<<"a="<<a<<endl; }
};

int main()
{ B op1;
  op1.getB(11);
  op1.dispB();
```

```

D1 op2;
op2.getD1(22);
op2.dispD1();
D2 op3;
op3.getD2(44);
op3.dispD2();
return 0;
}

```

编译上面的程序，在行尾标有“错误”的语句上产生了错误。原因是基类 B 中的保护成员 a 被其派生类 D1 私有继承后成为私有成员，所以不能被 D1 的派生类 D2 中的成员函数 dispD2 访问。

如果将例中成员函数 dispD2 改成如下形式：

```

void dispD2()
{ dispD1(); }

```

重新编译，程序将顺利通过。请读者想一想，为什么？

经修改后，程序的运行结果为：

```

a=11
a=22
a=44

```

2. 公有继承的访问规则

当类的继承方式为公有继承时，基类的公有成员和保护成员被继承到派生类中仍作为派生类的公有成员和保护成员，派生类的其他成员可以直接访问它们。但是，在类的外部只能通过派生类的对象访问继承来的公有成员，而不能访问继承来的保护成员。基类的私有成员在私有派生类中是不可直接访问的，所以无论是派生类成员函数，还是通过派生类的对象，都无法直接访问从基类继承来的私有成员，但是可以通过基类提供的公有成员函数间接访问它们。通过表 5.3 总结了公有继承的访问规则。

表 5.3 公有继承的访问规则

基类中的成员		私有成员	公有成员	保护成员
访问方式	内部访问 对象访问	不可访问 不可访问	可访问 可访问	可访问 不可访问

下面我们举一个公有继承的例子。

例 5.3 公有继承的示例。

```

#include<iostream>
using namespace std;
class B{ //声明一个基类 B
public:
    void getab(int a1, int b1)
    { a=a1; b=b1; }
    void dispab()
    { cout<<"a="<<a<<endl;
      cout<<"b="<<b<<endl;
    }
private:

```

```

    int a;
protected:
    int b;
};
class D: public B{           //声明一个公有派生类 D
public:
    void getabc(int a1, int b1, int c1)
    { getab(a1, b1);         //基类的函数 getab 在派生类中是公有成员
      c=c1;                  //派生类成员函数可以访问
    }
    void dispabc()
    { cout<<"a"<<a<<endl;    //错误, a 在派生类 D 中为不可直接访问成员
      cout<<"b"<<b<<endl;    //正确, b 在派生类 D 中为保护成员
      cout<<"c"<<c<<endl;    //派生类成员函数可以访问
    }
private:
    int c;
};
int main()
{ D obj;
  obj.getabc(11, 22, 33);
  obj.dispab();              //正确, 函数 dispab 在类 D 中为公有成员, 派生类对象能访问它
  obj.dispabc();
  return 0;
}

```

例中类 D 由类 B 公有派生出来, 所以类 B 中的两个公有成员函数 `getab` 和 `dispab` 在公有派生类中仍是公有成员。因此, 它们可以分别被派生类的成员函数 `getabc` 和派生类的对象 `obj` 访问。基类 B 中的数据成员 `a` 是私有成员, 它在派生类中是不能直接访问的, 所以在函数 `dispabc` 中对 `a` 的访问是非法的。基类 B 中的数据成员 `b` 是保护成员, 它在公有派生类中仍是保护成员, 所以在函数 `dispabc` 中对 `b` 的访问是合法的。

如果将例中成员函数 `dispabc()` 改成如下形式:

```

void dispabc()
{ dispab();
  cout<<"c"<<c<<endl;
}

```

重新编译, 程序将顺利通过。

修改后, 程序的运行结果如下:

```

a=11
b=22
a=11
b=22
c=33

```

说明:

需要再次强调, 派生类以公有继承的方式继承了基类, 并不意味着派生类可以访问基类的私有成员。如在例 5.3 的派生类的成员函数中, 企图访问基类 B 的私有成员 `a` 是非法的,

因为基类无论怎样被继承，它的私有成员对派生类而言都是不能直接访问的。

```
void dispabc()
{ cout<<"a="<<a<<endl;      //非法
  cout<<"b="<<b<<endl;
  cout<<"c="<<c<<endl;
}
```

3. 保护继承的访问规则

当类的继承方式为保护继承时，基类的公有成员和保护成员被继承到派生类中都作为派生类的保护成员，派生类的其他成员可以直接访问它们，但是在类的外部，不能通过派生类的对象来访问它们。基类的私有成员在私有派生类中是不可直接访问的，所以无论是派生类成员还是通过派生类的对象，都无法直接访问基类的私有成员。表 5.4 总结了保护继承的访问规则。

表 5.4 保护继承的访问规则

基类中的成员		私有成员	公有成员	保护成员
访问方式	内部访问 对象访问	不可访问 不可访问	可访问 不可访问	可访问 不可访问

例 5.4 保护继承的示例。

```
#include<iostream>
using namespace std;
class B{                      //声明基类 B
public:
    int z;
    void getx(int i)
    { x=i; }
    int putx()
    { return x; }
private:
    int x;
protected:
    int y;
};
class D: protected B{        //声明基类 B 的保护派生类 D
public:
    int p;
    void getall(int a, int b, int c, int d, int e, int f);
    void disp();
private:
    int m;
protected:
    int n;
};
void D::getall(int a, int b, int c, int d, int e, int f)
{ x=a;                        //错误，在派生类 D 中，x 为不可直接访问成员，可修改为"getx(a);"
  y=b;                        //合法，y 在类 D 中为保护成员
  z=c;                        //合法，z 在类 D 中为保护成员
  m=d;
```

```

    n=e;
    p=f;
}
void D::disp()
{ cout<<"x="<<x<<endl;          //错误, 在类 D 中, x 为不可直接访问成员
                                   //可修改为"cout<<"x="<<putx()<<endl;"

    cout<<"y="<<y<<endl;          //合法, y 在类 D 中为保护成员
    cout<<"z="<<z<<endl;          //合法, z 在类 D 中为保护成员
    cout<<"m="<<m<<endl;
    cout<<"n="<<n<<endl;
}
int main()
{ D obj;
  obj.getall(1, 2, 3, 4, 5, 6);
  obj.disp();
  cout<<"p="<<obj.p<<endl;        //合法, p 在类 D 中为公有成员
  return 0;
}

```

将类 D 的函数 getall 中的语句“x=a;”修改为“getx(a);”, 函数 disp 中的语句“cout<<"x="<<x<<endl;”修改为“cout<<"x="<<putx()<<endl;”后, 本程序的运行结果为:

```

x=1
y=2
z=3
m=4
n=5
p=6

```

5.2 派生类的构造函数和析构函数

构造函数的主要作用是对数据初始化。在派生类中, 如果对派生类新增的成员进行初始化, 就需要加入派生类的构造函数。与此同时, 对所有从基类继承下来的成员的初始化工作, 还是由基类的构造函数完成, 但是基类的构造函数和析构函数不能被继承, 因此我们必须在派生类的构造函数中对基类的构造函数所需要的参数进行设置。同样, 对撤销派生类对象时的扫尾、清理工作也需要加入新的析构函数来完成。这些都是本节所要讨论的问题。

5.2.1 派生类构造函数和析构函数的调用顺序

通常情况下, 当创建派生类对象时, 首先调用基类的构造函数, 随后再调用派生类的构造函数; 当撤销派生类对象时, 则先调用派生类的析构函数, 随后再调用基类的析构函数。

下列程序的运行结果, 反映了基类和派生类的构造函数及析构函数的调用顺序。

例 5.5 基类和派生类的构造函数及析构函数的调用顺序。

```

#include<iostream>
using namespace std;

```

```

class B{                //声明基类 B
public:
    B()                //基类的构造函数
    { cout<<"Constructor of B\n"; }
    ~B()               //基类的析构函数
    { cout<<"Destructor of B\n"; }
};
class D:public B {      //声明基类 B 的公有派生类 D
public:
    D()                //派生类的构造函数
    { cout<<"Constructor of D\n"; }
    ~D()               //派生类的析构函数
    { cout<<"Destructor of D\n"; }
};
int main()
{ D op;
  return 0;
}

```

程序运行结果如下:

```

Constructor of B
Constructor of D
Destructor of D
Destructor of B

```

从程序运行的结果可以看出: 构造函数的调用严格地按照先调用基类的构造函数, 后调用派生类的构造函数的顺序执行。析构函数的调用顺序与构造函数的调用顺序正好相反, 先调用派生类的析构函数, 后调用基类的析构函数。

5.2.2 派生类构造函数和析构函数的构造规则

1. 简单的派生类的构造函数和析构函数

简单的派生类只有一个基类, 而且只有一级派生 (只有直接派生类, 没有间接派生类), 在派生类的成员数据中不包含基类的对象 (即子对象)。下面先介绍在简单的派生类中怎样定义构造函数。

当基类的构造函数没有参数, 或没有显式定义构造函数时, 派生类可以不向基类传递参数, 甚至可以不定义构造函数。例 5.5 的程序就是由于基类的构造函数没有参数, 所以派生类没有向基类传递参数。

派生类不能继承基类中的构造函数和析构函数。当基类含有带参数的构造函数时, 派生类必须定义构造函数, 以提供把参数传递给基类构造函数的途径。

在 C++ 中, 派生类构造函数的一般格式为:

派生类名(参数总表):基类名(参数表)

```

{
    派生类新增数据成员的初始化语句
}

```

其中基类构造函数的参数, 通常来源于派生类构造函数的参数总表, 也可以用常数值。

在派生类中可以根据需要定义自己的析构函数，用来对派生类中的所增加的成员进行清理工作。基类的清理工作仍然由基类的析构函数负责。由于析构函数是不带参数的，在派生类中是否要自定义析构函数与它所属基类的析构函数无关。在执行派生类的析构函数时，系统会自动调用基类的析构函数，对基类的对象进行清理。

下面的程序说明如何传递一个参数给派生类的构造函数和传递一个参数给基类的构造函数，以及派生类析构函数的定义方法。

例 5.6 当基类含有带参数的构造函数时，派生类构造函数和析构函数的构造方法。

```
#include<iostream>
using namespace std;
class B{                                //声明基类 B
public:
    B(int n)                            //基类的构造函数
    { cout<<"Constructor of B"<<endl;
      i=n;
    }
    ~B()                                //基类的析构函数
    { cout<<"Destructor of B"<<endl; }
    void disp1()
    { cout<<i<<endl; }
private:
    int i;
};
class D :public B{                      //声明基类 B 的公有派生类 D
public:
    D(int n, int m):B(m)                //定义派生类构造函数时
    {                                   //写上要调用的基类构造函数及其参数
        cout<<"Constructor of D"<<endl;
        j=n;
    }
    ~D()                                //派生类的析构函数
    { cout<<"Destructor of D"<<endl; }
    void dispj()
    { cout<<j<<endl; }
private:
    int j;
};
int main()
{ D obj(50, 60);
  obj.disp1();
  obj.dispj();
  return 0;
}
```

程序运行结果如下：

Constructor of B

Constructor of D

60

50

Destructor of D

Destructor of B

2. 含有子对象的派生类的构造函数

前面介绍过的派生类,其数据成员都是基本数据类型(如 int, char)或系统提供的类型(如 string)。实际上,派生类的数据成员还可以是基类的对象,称为子对象,即对象中的对象。

当派生类中含有子对象时,其构造函数的一般形式为:

派生类名(参数总表):基类名(参数表 0), 子对象名 1(参数表 1), ...,
子对象名 n(参数表 n)

```
{
    派生类新增成员的初始化语句
}
```

在定义派生类对象时,构造函数的调用顺序如下:

- 调用基类的构造函数,对基类数据成员初始化;
- 调用子对象的构造函数,对子对象的数据成员初始化;
- 调用派生类的构造函数体,对派生类数据成员初始化。

撤销对象时,析构函数的调用顺序与构造函数的调用顺序正好相反。首先调用派生类的析构函数,然后调用子对象的析构函数,最后调用基类的析构函数。

下面这个程序说明派生类中内嵌子对象时派生类构造函数和析构函数的调用顺序。

例 5.7 内嵌子对象时派生类构造函数和析构函数的调用顺序。

```
#include<iostream>
using namespace std;
class B{                                //声明基类 B
public:
    B(int i)                            //基类的构造函数
    { x=i;
      cout<<"Constructor of B\n"; }
    ~B()                                //基类的析构函数
    { cout<<"Destructor of B\n"; }
    void show()
    { cout<<"x=" <<x<<endl; }
private:
    int x;
};
class D:public B{                       //声明公有派生类 D
public:
    D(int i):B(i), d(i)                 //派生类的构造函数,继上要调用的
    { cout<<"Constructor of D\n"; }     //基类构造函数和子对象构造函数
    ~D()                                 //派生类的析构函数
    { cout<<"Destructor of D\n"; }
private:
    B d;                                //定义子对象 d
};
int main()
{ D obj(456);
  obj.show();
```

```

    return 0;
}

```

程序执行结果如下:

```

Constructor of B

```

```

Constructor of B

```

```

Constructor of D

```

```

x=456

```

```

Destructor of D

```

```

Destructor of B

```

```

Destructor of B

```

上面程序中有两个类, 基类 B 和派生类 D。基类中含有一个需要传递参数的构造函数, 用它初始化私有成员 x, 并显示出一句信息。派生类 D 中含有子对象 d。从程序执行的结果分析, 构造函数和析构函数的调用顺序与规定的顺序是完全一致的。

说明:

(1) 当基类构造函数不带参数时, 派生类不一定需要定义构造函数, 然而当基类的构造函数哪怕只带有一个参数, 它所有的派生类都必须定义构造函数。甚至所定义的派生类构造函数的函数体可能为空, 仅仅起参数的传递作用。

例如, 在下面的程序段中, 派生类 D 就不使用参数 n, n 只是被传递给了要调用的基类构造函数 B。

```

class B{
    int i;
public:
    B(int n)
    { cout<<"Constructor of B\n";
      i=n;
    }
    void showi()
    { cout<<i<<"\n"; }
};

class D:public B{
    int j;
public:
    D(int n):B(n)
    { cout<<"Constructor of D\n";
      j=0;
    }
    void showj()
    { cout<<j<<"\n"; }
};

```

(2) 若基类使用默认构造函数或不带参数的构造函数, 则在派生类中定义构造函数时可略去“:基类名(参数表)”, 此时若派生类也不需要构造函数, 则可不定义构造函数。

(3) 如果派生类的基类也是一个派生类, 每个派生类只需负责其直接基类数据成员的初始化, 依次上溯。

5.3 在派生类中显式访问基类成员

在定义派生类的时候，C++语言允许在派生类中说明的成员与基类中说明的成员名字相同，也就是说，派生类可以重新说明与基类成员同名的成员。在没有虚函数的情况下（虚函数在第6章介绍），如果在派生类中定义了与基类成员同名的成员，则称派生类成员覆盖了基类的同名成员，在派生类中使用这个名字意味着访问在派生类中重新说明的成员。为了在派生类中使用基类的同名成员，必须在该成员名之前加上基类名和作用域标识符“::”，即必须使用下列格式才能访问到基类的同名成员。

基类名::成员名

或

基类名:: 成员函数(参数表);

下面的程序片段说明了这个要点：

```
class X{
public:
    int f();
};
class Y:public X{
public:
    int f();
    int g();
};
void Y::g()
{ f();                                     //表示访问派生类中的 f(), 即被调用的函数是 Y::f()
}                                           //若要访问基类中的 f(), 应改写成 X::f()
```

对于派生类对象的访问，也有相同的结论。例如：

```
Y obj;
obj.f();                                   //被访问的函数是 Y::f()
```

如果要访问基类中声明的同名成员，则应使用作用域标识符限定，例如：

```
obj.X::f();                               //被调用的函数是 X::f()
```

例 5.8 在派生类中定义同名成员。

```
#include<iostream>
#include<string>
using namespace std;
class Student{                               //声明基类 Student
public:
    Student(int number1, string name1, float score1) //基类构造函数
    { number=number1;
      name=name1;
      score=score1;
    }
};
```

```

        void print()                                //在基类中定义了成员函数 print
    { cout<<"number:"<<number<<endl;
      cout<<"name:"<<name<<endl;
      cout<<"score:"<<score<<endl;
    }
protected:
    int number;                                     //学号
    string name;                                    //姓名
    float score;                                    //成绩
};
class UStudent:public Student{                      //声明公有派生类 UStudent
public:
    UStudent(int number1, string name1, float score1, string major1)
    :Student(number1, name1, score1)                //定义派生类构造函数时
    { major=major1;                                //继上对基类的构造函数的调用
    }
    void print()                                    //在派生类中定义了同名的成员函数 print
    { Student::print();                             //显式调用基类 Student 的成员函数 print
      cout<<"major:"<<major<<endl;
    }
private:
    string major;                                   //专业
};
int main()
{ UStudent stu(22116, "张志", 95, "信息安全");
  stu.print();                                     //调用的是派生类中的成员函数 print
  return 0;
}

```

在本例的基类 `Student` 中定义了成员函数 `print`，在派生类 `UStudent` 中，定义了同名成员函数 `print`。在主程序中派生类对象 `stu` 调用的是派生类中的成员函数 `print`。若要在派生类的成员函数 `print` 中调用基类的成员函数 `print`，必须在该成员名之前加上基类名和作用域标识符 `::`，即 `Student::`。在面向对象程序设计中，若要在派生类中对基类继承过来的某些函数功能进行扩充和改造，都可以通过这样的覆盖来实现。这种覆盖的方法，是对基类成员改造的关键手段，是程序设计中经常使用的方法。

本程序的运行结果如下：

```

number:22116
name:张志
score:95
major:信息安全

```

5.4 多重继承与虚基类

前面介绍的是单继承，即一个类是从一个基类派生而来的。实际上，常常有这样的情况：一个派生类有两个或多个基类，派生类从两个或多个基类中继承所需的属性。例如，计算机

屏幕上用户界面所提供的窗口、滚动条、文本框以及多种类型的按钮，所有这些组件都是通过类来支持的。若把这些类中的两个类或多个类合并，则可产生一个新类，例如把窗口类和滚条类合并起来产生一个可滚动的窗口类，这个可滚动的窗口类就是从多个基类继承而来的。当一个派生类具有多个基类时，这种派生方法称为多基派生或多重继承。

5.4.1 声明多重继承派生类的方法

在 C++ 中，声明具有多个基类的派生类与声明单基派生类的形式相似，只需将要继承的多个基类使用逗号分隔即可。例如已经声明了类 X 和类 Y。可以声明多重继承的派生类 Z：

```
class Z:public X, private Y{           //类 Z 公有继承了类 X，私有继承了类 Y
```

派生类 Z 中新增的数据成员和成员函数

```
};
```

声明多重继承派生类的一般形式如下：

```
class 派生类名:继承方式1 基类名1, ..., 继承方式i 基类名i, ...,
```

继承方式 n 基类名 n

```
{
```

派生类新增的数据成员和成员函数

```
};
```

冒号后面的部分称为基类表，各基类之间用逗号分隔，其中“继承方式 i” (i=1, 2, ..., n) 规定了派生类从基类中按什么方式继承:private、protected 或 public。默认的继承方式是 private。例如：

```
class Z:X, public Y{                  //类 Z 私有继承了类 X，公有继承了类 Y
```

```
//...
```

```
};
```

```
class Z:public X, public Y{          //类 Z 公有继承了类 X 和类 Y
```

```
//...
```

```
};
```

在多重继承中，公有继承和私有继承对于基类成员在派生类中的访问属性与单继承的规则相同。

下面程序中类 C 继承了类 A 和类 B，请注意各成员的访问特性有什么变化。

例 5.9 多重继承情况下的访问特性。

```
#include<iostream>
using namespace std;
class A{                               //声明基类 A
public:
    void setA(int x)
    { a=x; }
    void printA()
    { cout<<"a="<<a<<endl; }
private:
    int a;
};
class B{                               //声明基类 B
public:
    void setB(int x)
```

```

    { b=x; }
    void printB()
    { cout<<"b="<<b<<endl; }
private:
    int b;
};

class C:public A, private B { //声明派生类 C, 公有继承了类 A, 私有继承了类 B
public:
    void setC(int x, int y)
    { c=x;
      setB(y); //成员函数 setB 在 C 中为私有成员
    }
    void printC()
    { printB(); //成员函数 printB 在 C 中为私有成员
      cout<<"c="<<c<<endl;
    }
private:
    int c;
};

int main()
{ C obj;
  obj.setA(11); //正确, 成员函数 setA 在 C 中仍是公有成员
  obj.printA(); //正确, 成员函数 printA 在 C 中仍是公有成员
  obj.setB(33); //错误, 成员函数 setB 在 C 中已成为私有成员
  obj.printB(); //错误, 成员函数 printB 在 C 中已成为私有成员
  obj.setC(55, 88); //正确, 成员函数 setC 在 C 中是公有成员
  obj.printC(); //正确, 成员函数 printC 在 C 中公有成员
  return 0;
}

```

在上面的程序中, 类 A 和类 B 是两个基类, 类 C 是从类 A 和类 B 派生出来的。从派生方式可以看到, 类 C 从类 A 公有派生和从类 B 私有派生出来。根据派生的有关规则, 类 A 的公有成员在 C 中仍是公有成员, 类 B 的公有成员在 C 中成为私有成员。所以, 在主函数中对类 A 的公有成员函数 setA 的访问是正确的, 因为在 C 中它仍是公有成员; 对 B 的成员函数 setB 的访问是错误的, 因为 B 的成员函数 setB 在 C 中已成为私有成员, 不能直接访问。

删去标有错误的两条语句, 程序运行结果如下:

```

a=11
b=88
c=55

```

第 1 行的输出结果是“obj.printA();”产生的。第 2 行和第 3 行的输出结果是“obj.printC();”产生的, 其中第 2 行的输出结果是函数 printC 调用函数 printB 产生的。

说明:

对基类成员的访问必须是无二义性的, 例如下列程序段对基类成员的访问是二义性的, 必须想法消除二义性。

```

class X{
public:
    int f();
};

```

```

class Y{
public:
    int f();
    int g();
};
class Z:public X, public Y{
public:
    int g();
    int h();
};

```

如定义类 Z 的对象 obj:

```
Z obj;
```

则以下对函数 f() 的访问是二义的:

```
obj.f(); //二义性错误, 不知调用的是类 X 的 f(), 还是类 Y 的 f()。
```

使用成员名限定可以消除二义性, 例如:

```

obj.X::f(); //调用类 X 的 f()
obj.Y::f(); //调用类 Y 的 f()

```

5.4.2 多重继承派生类的构造函数与析构函数

多重继承派生类构造函数的定义形式与单继承时的构造函数定义形式相似, 只是在初始表中包含多个基类构造函数。多个基类的构造函数之间用“,”分隔。多重继承构造函数定义的一般形式如下:

派生类名(参数总表):基类名 1(参数表 1), ..., 基类名 i(参数表 i), ..., 基类名 n(参数表 n)

```

{
    派生类新增成员的初始化语句
}

```

与单继承下派生类构造函数相同, 多重继承下派生类构造函数必须同时负责该派生类所有基类构造函数的调用。

多重继承构造函数的调用顺序与单继承构造函数的调用顺序相同, 也是遵循先调用基类的构造函数, 再调用对象成员的构造函数, 最后调用派生类构造函数的原则。析构函数的调用顺序则刚好与构造函数的调用顺序相反。

下面我们再看一个程序, 其中类 X 和类 Y 是基类, 类 Z 是类 X 和类 Y 共同派生出来的, 请注意类 Z 的构造函数的定义方法。

例 5.10 多重继承情况下派生类构造函数和析构函数的定义方法。

```

#include<iostream>
using namespace std;
class X{

public:
    X(int sa) //基类 X 的构造函数
    { a=sa; }
    int getX()
    { return a; }
    ~X() //基类 X 的析构函数
}

```

```

    { cout<<"Destructor of X called."<<endl; }
private:
    int a;
};
class Y{
public:
    Y(int sb) //基类 Y 的构造函数
    { b=sb;}
    int getY()
    { return b;}
    ~Y() //基类 Y 的析构函数
    { cout<<"Destructor of Y called."<<endl; }
private:
    int b;
};
class Z:public X, private Y{ //类 Z 为基类 X 和基类 Y 共同的派生类
public:
    Z(int sa, int sb, int sc):X(sa), Y(sb) //派生类 Z 的构造函数, 继上了对
    { c=sc; } //基类 X 和 Y 的构造函数的调用
    int getZ()
    { return c;}
    int getY()
    { return Y::getY();}
    ~Z() //派生类 Z 的析构函数
    { cout<<"Destructor of Z called."<<endl; }
private:
    int c;
};
int main()
{ Z obj(222, 444, 666);
  int ma=obj.getX();
  cout<<"ma="<<ma<<endl;
  int mb=obj.getY();
  cout<<"mb="<<mb<<endl;
  int mc=obj.getZ();
  cout<<"mc="<<mc<<endl;
  return 0;
}

```

在上述程序中, 定义派生类 Z 的构造函数时, 它的参数表中给出了初始化对象时所需要的参数 sa、sb 和 sc。冒号后面列出了基类 X 和基类 Y 的构造函数, 并指出把 sa 传递给基类 X 的构造函数, 把 sb 传递给基类 Y 的构造函数。这样, 在创建类 Z 的对象时, 它的构造函数就会自动地用参数表中的数据调用基类的构造函数, 完成基类对象的初始化。

在主函数 main 中创建了类 Z 的一个对象 obj, 并用 222、444 和 666 这 3 个参数初始化 obj 的数据成员。这 3 个参数传递给对象 obj 的构造函数 obj.Z(int sa, int sb, int sc), 这个构造函数用参数 sa 调用基类 X 的构造函数 X(int sa), 由 X(int sa) 把 sa 的值赋给 a, 然后用参数 sb 调用基类 Y 的构造函数 Y(int sb), 由 Y(int sb) 把 sb 的值赋给 b, 最后把 sc 的值赋给 c, 初始化的过程就完成了。

由于派生类 Z 是 X 公有派生出来的, 所以类 X 中的公有成员函数 getX 在类 Z 中仍是公有的, 在 main 中可以直接引用, 把成员 a 的值赋给主函数 main 中的变量 ma, 并显示在屏幕

上。类 Z 是从类 Y 私有派生出来的, 所以类 Y 中的公有成员函数 getY 在类 Z 中成为私有的, 在主函数 main 中不能直接引用。为了能取出 b 的值, 在 Z 中另外定义了一个公有成员函数 Z::getY(), 它通过调用 Y::getY() 取出 b 的值。主函数 main 中的语句:

```
int mb=obj.getY();
```

调用的是派生类 Z 的成员函数 getY, 而不是基类 Y 的成员函数 getY。由于类 Z 中的成员函数 getZ 是公有成员, 所以在主函数 main 中可以直接调用取出 c 的值。上述程序运行的结果如下:

```
ma=222
mb=444
mc=666
Destructor of Z called.
Destructor of Y called.
Destructor of X called.
```

由于析构函数是不带参数的, 在派生类中是否要定义析构函数与它所属的基类无关, 所以与单继承情况类似, 基类的析构函数不会因为派生类没有析构函数而得不到执行, 它们各自是独立的。析构函数的调用顺序则刚好与构造函数的调用顺序相反。

说明:

多重继承构造函数的调用顺序是: 先调用基类的构造函数, 再调用对象成员的构造函数, 最后调用派生类构造函数的原则。处于同一层次的各个基类构造函数的调用顺序, 取决于声明派生类时所指定的各个基类的顺序, 与派生类构造函数中所定义的成员初始化列表的各项顺序没有关系。

例 5.11 处于同一层次的基类构造函数的调用顺序。

```
#include<iostream>
using namespace std;
class B1{
public:
    B1(int i)
    { b1=i;
      cout<<"Constructor B1. "<<endl;
    }
    void Print()
    { cout<<b1<<endl;
    }
private:
    int b1;
};
class B2{
public:
    B2(int i)
    { b2=i;
      cout<<"Constructor B2. "<<endl;
    }
    void Print()
    { cout<<b2<<endl;
    }
private:
```

```

    int b2;
};
class A:public B2, public B1{    //声明派生类时，基类的顺序是：先基类 B2，后基类 B1
public:
    A(int i, int j, int l);
    void Print();
private:
    int a;
};
A::A(int i, int j, int l):B1(i), B2(j)
{ a=l;
  cout<<"Constructor A. "<<endl;
}
void A::Print()
{ B1::Print();
  B2::Print(); cout<<a<<endl;
}
int main()
{ A aa(3, 2, 1);
  aa.Print();
  return 0;
}

```

本例在声明派生类 A 时，基类的顺序是：先基类 B2，后基类 B1。所以，当定义派生类 A 的对象 aa 时，构造函数的调用顺序是：先调用基类 B2 的构造函数，再调用基类 B1 的构造函数，最后调用派生类 A 的构造函数。

本程序的运行结果如下：

```

Constructor B2.
Constructor B1.
Constructor A.
3
2
1

```

5.4.3 虚基类

1. 虚基类的作用

如果一个类有多个直接基类，而这些直接基类又有一个共同的基类，则在最低层的派生类中会保留这个间接的共同基类数据成员的多份同名成员。在派生类中访问这些同名的成员时，必须加上“基类名::”，使其唯一地标识是哪一个基类的成员，以免产生二义性。请看下面的例题。

例 5.12 虚基类的引例。

```

#include<iostream>
using namespace std;
class Base{                                //声明类 Base1 和类 Base2 共同的基类 Base
public:
    Base()
    { a=5;

```

```

        cout<<"Base a="<<a<<endl;
    }
protected:
    int a;
};
class Base1:public Base{                               //声明 Base1 是 Base 的派生类
public:
    Base1()
    { a=a+10;
      cout<<"Base1 a="<<a<<endl;                      //这是类 Base1 的 a, 即 Base1::a
    }
};
class Base2:public Base{                               //声明 Base2 是 Base 的派生类
public:
    Base2()
    { a=a+20;
      cout<<"Base2 a="<<a<<endl;                      //这是类 Base2 的 a, 即 Base2::a
    }
};
class Derived:public Base1, public Base2{              //Derived 是 Base1 和 Base2 的共同派生类, 是 Base 的间接派生类
public:
    Derived()
    { cout<<"Base1::a="<<Base1::a<<endl;              //在 a 前面加上 "Base1::"
      cout<<"Base2::a="<<Base2::a<<endl;              //在 a 前面加上 "Base2::"
    }
};
int main()
{ Derived obj;
  return 0;
}

```

程序运行结果如下:

```

Base a=5
Base1 a=15
Base a=5
Base2 a=25
Base::a=15
Base2::a=25

```

上述程序中, 类 `Derived` 是从类 `Base1` 和 `Base2` 公有派生而来, 而类 `Base1` 和类 `Base2` 又都是从类 `Base` 公有派生而来的。虽然在类 `Base1` 和类 `Base2` 中没有定义数据成员 `a`, 但是它们分别从类 `Base` 继承了数据成员 `a`, 这样在类 `Base1` 和类 `Base2` 中同时存在着同名的数据成员 `a`, 它们都是类 `Base` 成员的拷贝。但是类 `Base1` 和类 `Base2` 中的数据成员 `a` 分别具有不同的存储单元, 可以存放不同的数据。在程序中可以通过类 `Base1` 和类 `Base2` 调用基类 `Base` 的构造函数, 分别对类 `Base1` 和类 `Base2` 的数据成员 `a` 初始化。图 5.2 表示了这个例子中类之间的层次关系。

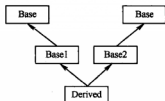


图 5.2 例 5.12 的类层次图

由于在类 `Derived` 中同时存在着类 `Base1` 和类 `Base2` 的数据成员 `a`，因此在 `Derived` 的构造函数中输出 `a` 的值，必须加上“类名::”，指出是哪一个数据成员 `a`，否则就会出现二义性。如果将例 5.12 中的派生类 `Derived` 改成以下形式：

```
class Derived:public Base1, public Base2{
public:
    Derived()
    { cout<<"Derived a="<<a<<endl; //错误，存在二义性
    }
```

运行这个程序将出现错误，问题就出在派生类 `Derived` 的构造函数的定义上，它试图输出数据成员 `a` 的值，表面上看来这是合理的，但实际上这时对 `a` 的访问存在二义性，即类中的数据成员 `a` 的值可能是从 `Base1` 的派生路径上来的 `Base1::a`，也有可能是从类 `Base2` 的派生路径上来的 `Base2::a`，这里没有明确的说明。

为了解决这种二义性，C++引入了虚基类的概念。

2. 虚基类的声明

不难理解，如果在上例中类 `base` 只存在一个拷贝（即只有一个数据成员 `a`），那么对 `a` 的访问就不会产生二义性。在 C++中，可以通过将这个公共的基类说明为虚基类来解决这个问题。这就要求从类 `base` 派生新类时，使用关键字 `virtual` 将类 `base` 说明为虚基类。

声明虚基类的语法形式如下：

```
class 派生类名:virtual 继承方式 类名{
...
}
```

下面我们用虚基类重新声明例 5.12 中的类。

例 5.13 虚基类的声明。

```
#include<iostream>
using namespace std;

class Base{ //声明基类 Base
public:
    Base()
    { a=5;
      cout<<"Base a="<<a<<endl;
    }
protected:
    int a;
};

class Base1: virtual public Base{ //声明类 Base 是 Base1 的虚基类
public:
    Base1()
    { a=a+10;
      cout<<"Base1 a="<<a<<endl;
    }
};

class Base2: virtual public Base{ //声明类 Base 是 Base2 的虚基类
public:
    Base2()
    { a=a+20;
      cout<<"Base2 a="<<a<<endl;
    }
};
```

```

    }
};
class Derived:public Base1, public Base2{
    //Derived是 Base1 和 Base2 的共同派生类, 是 Base 的间接派生类
public:
    Derived()
    { cout<<"Derived a="<<a<<endl; }
};
int main()
{ Derived obj;
  return 0;
}

```

程序运行结果如下:

```

Base a=5
Base1 a=15
Base2 a=35
Derived a=35

```

在上述程序中, 从类 Base 派生出类 Base1 和类 Base2 时, 使用了关键字 virtual, 把类 Base 声明为 Base1 和 Base2 的虚基类。这样, 从 Base1 和 Base2 派生出的类 Derived 只继承基类 Base 一次, 也就是说, 基类 Base 的数据成员 a 只保留一份。当在派生类 Base1 和 Base2 中作了以上的虚基类声明后, 这个例子中类之间的层次关系如图 5.3 所示。

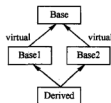


图 5.3 例 5.13 的类层次图

说明:

关键字 virtual 与继承方式关键字 (public 或 private) 的先后顺序无关紧要, 它只说明是“虚拟继承”。例如以下两个虚拟继承的声明是等价的。

```

class Dderived:virtual public Base{
    ...
};
class Derived:public virtual Base{
    ...
};

```

3. 虚基类的初始化

虚基类的初始化与一般的多重继承的初始化在语法上是一样的, 但构造函数的调用顺序不同。在使用虚基类机制时应该注意以下几点。

(1) 如果在虚基类中定义有带形参的构造函数, 并且没有定义默认形式的构造函数, 则在整个继承结构中, 所有直接或间接的派生类都必须在构造函数的成员初始化表中列出对虚基类构造函数的调用, 以初始化在虚基类中定义的数据成员。

(2) 建立一个对象时, 如果这个对象中含有从虚基类继承来的成员, 则虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。该派生类的其他基类对虚基类构造函数的调用都自动被忽略。

(3) 若同一层次中同时包含虚基类和非虚基类, 应先调用虚基类的构造函数, 再调用非虚基类的构造函数, 最后调用派生类构造函数。

(4) 对于多个虚基类, 构造函数的执行顺序仍然是先左后右, 自上而下。

(5) 对于非虚基类, 构造函数的执行顺序仍是先左后右, 自上而下。

(6) 若虚基类由非虚基类派生而来, 则仍然先调用基类构造函数, 再调用派生类的构造函数。例如:

```
class X:public Y, virtual public Z{
    ...
};
X one;
```

定义类 X 的对象 one 后, 将产生如下的调用次序。

```
Z();
Y();
X();
```

下面的程序说明了含有虚基类的派生类构造函数的执行顺序。

例 5.14 含有虚基类的派生类构造函数的执行顺序。

```
#include<iostream>
using namespace std;
class Base { //声明基类 Base
public:
    Base(int sa)
    { a=sa;
      cout<<"Constructing Base"<<endl;
    }
private:
    int a;
};
class Base1:virtual public Base{ //声明类 Base 是 Base1 的虚基类
public:
    Base1(int sa, int sb):Base(sa) //在此, 必须缀上对类 Base 构造函数的调用
    { b=sb;
      cout<<"Constructing baes1"<<endl;
    }
private:
    int b;
};
class Base2:virtual public Base{ //声明类 Base 是 Base2 的虚基类
public:
    Base2(int sa, int sc):Base(sa) //在此, 必须缀上对类 Base 构造函数的调用
    { c=sc;
      cout<<"Constructing baes2"<<endl;
    }
private:
    int c;
};
class Derived:public Base1, public Base2
{ //Derived 是 Base1 和 Base2 的共同派生类, 是 Base 的间接派生类
public:
    Derived(int sa, int sb, int sc, int sd):
        Base(sa), Base1(sa, sb), Base2(sa, sc)
    { //在此, 必须缀上对类 Base 构造函数的调用
```

```

        d=sd;
        cout<<"Constructing Derived"<<endl;
    }
private:
    int d;
};
int main()
{ Derived obj(2, 4, 6, 8);
  return 0;
}

```

在上述程序中，Base 是一个虚基类，它只有一个带参数的构造函数，因此要求在派生类 Base1、Base2 和 Derived 的构造函数的初始化表中，都必须带有对类 Base 构造函数的调用。如果 Base 不是虚基类，在派生类 Derived 的构造函数的初始化表中调用类 Base 的构造函数是错误的，但是当 Base 是虚基类且只有带参数的构造函数时，就必须在类 Derived 的构造函数的初始化表中调用类 Base 的构造函数。因此，在类 Derived 构造函数的初始化表中，不仅含有对类 Base1 和类 Base2 构造函数的调用，还有对虚基类 Base 构造函数的调用。

上述程序运行的结果如下：

```

Constructing Base
Constructing base1
Constructing base2
Constructing Derived

```

不难看出，上述程序中虚基类 Base 的构造函数只执行了一次。显然，当 Derived 的构造函数调用了虚基类 Base 的构造函数之后，类 Base1 和类 Base2 对 Base 构造函数的调用被忽略了。这也是初始化虚基类和初始化非虚基类不同的地方。

5.5 应用举例

例 5.15 声明一个共同的基类 Person，它包含了所有派生类共有的数据，职工类 Employee 和大学生类 Student 为虚基类 Person 的派生类，在职大学生类 E_student 是职工类 Employee 的派生类。每类人员具有的数据如下：

职工类 Employee：姓名、性别、年龄、部门；
 大学生类 Student：姓名、性别、年龄、专业；
 在职大学生类 E_student：姓名、性别、年龄、部门、导师。

每个类定义了一个相对于特定类的不同的 print 函数，输出各类的数据成员。类之间的关系如图 5.6 所示。

下面是具体的程序。

```

#include<iostream>
#include<string>
using namespace std;
class Person{
public:

```

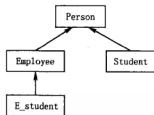


图 5.4 例 5.14 的类层次图

//声明基类 Person

```

    Person(string name1, char sex1, int age1)           //构造函数
    { name=name1;
      sex=sex1;
      age=age1;
    }
    void print()
    { cout<<"姓名 : "<< name<<endl;
      cout<<"性别 : "<< sex<<endl;
      cout<<"年龄 : "<< age<<endl;
    }
protected:
    string name;           //姓名
    char sex;              //性别
    int age;               //年龄
};

class Student:virtual public Person {                 //声明类 Person 是类 Student 的虚基类
public:
    Student(string name1, char sex1, int age1, string major1):    //构造函数
    Person(name1, sex1, age1)
    { major= major1; }
    void print()
    { Person:: print();
      cout<<"专业 : "<<major<<endl;
    }
protected:
    string major;         //专业
};

class Employee:virtual public Person{                 //声明类 Person 是类 Employee 的虚基类
public:
    Employee(string name1, char sex1, int age1, string dept1):    //构造函数
    Person(name1, sex1, age1)
    { dept=dept1;}
    void print()
    { Person:: print();
      cout<<"部门 : "<<dept<<endl;
    }
protected:
    string dept;          //部门
};

class E_Student:public Employee{                     //声明类 E_Student 为类 Employee 的派生类
public:
    E_Student(string name1, char sex1, int age1,           //构造函数
    string dept1, string tname1):
    Person(name1, sex1, age1), Employee(name1, sex1, age1, dept1)
    { tname=tname1;}
    void print()
    { Employee::print();
      cout<<"导师 : "<< tname<<endl;
    }
private:

```



```

        string tname;                                //导师
    };
    int main()
    { Student my_Student("李淑兰", 'f', 22, "应用数学");
      cout<<"大学生: " <<endl;
      my_Student.print();
      Employee my_Employee("黄福良", 'm', 55, "科研处");
      cout<<"职工: " <<endl;
      my_Employee.print();
      E_Student my_E_Student("张海燕", 'f', 35, "教务处", "吉秋石");
      cout<<"在职大学生:" <<endl;
      my_E_Student.print();
      return 0;
    }

```

程序运行的结果如下:

```

大学生:
姓名: 李淑兰
性别: f
年龄: 22
专业: 应用数学
职工:
姓名: 黄福良
性别: m
年龄: 55
部门: 科研处
在职大学生:
姓名: 张海燕
性别: f
年龄: 35
部门: 教务处
导师: 吉秋石

```

实 验

实验目的和要求

1. 学习派生类的声明方法和派生类构造函数的定义方法。
2. 学习在不同继承方式下, 基类成员在派生类中的访问属性。
3. 学习在继承方式下, 构造函数与析构函数的执行顺序与构造规则。
4. 学习虚基类在解决二义性问题中的作用。

实验内容和步骤

1. 编写一个学生和教师的数据输入和显示程序。学生数据有编号、姓名、性别、年龄、

系别和成绩,教师数据有编号、姓名、性别、年龄、职称和部门。要求将编号、姓名、性别、年龄的输入和显示设计成一个类 Person,并作为学生类 Student 和教师类 Teacher 的基类。

供参考的类结构如下:

```
class Person{
    ...
};
class Student: public Person{
    ...
};
class Teacher: public Person{
    ...
};
```

2. 按要求阅读、编辑、编译、调试和运行以下程序。

(1) 编辑、编译、调试和运行程序 tes5_2_1.cpp, 并写出程序的运行结果。

```
//test5_2_1.cpp
#include<iostream>
#include<string>
using namespace std;
class MyArray { //声明一个基类 MyArray
public:
    MyArray(int leng); //构造函数
    ~MyArray(); //析构函数
    void Input(); //输入数据的成员函数
    void Display(string); //输出数据的成员函数
protected:
    int *alist; //基类中存放一组整数
    int length; //整数的个数
};
MyArray::MyArray(int leng)
{ if (leng<=0)
{ cout<<"error length";
  exit(1);
}
  alist=new int [leng];
  length=leng;
  if (alist==NULL)
  { cout<<"assign failure";
    exit(1);
  }
  cout<<"MyArray 类对象已创建."<<endl;
}
MyArray::~MyArray()
{ delete[] alist;
  cout<<"MyArray 类对象被撤销."<<endl;
}
void MyArray::Display(string str) //显示数组内容
{ int i;
  int *p=alist;
```

```

    cout<<str<<length <<"个整数: ";
    for (i=0;i<length;i++, p++)
        cout<<*p<<" ";
    cout<<endl;
}

void MyArray::Input() //从键盘输入若干整数
{ cout<<"请从键盘输入"<<length <<"个整数: ";
  int i;
  int *p=alist;
  for (i=0;i<length;i++, p++)
      cin>>*p;
}

int main()
{ MyArray a(5);
  a.Input(); //输入整数
  a.Display("显示已输入的"); //显示已输入的整数
  return 0;
}

```

(2) 声明一个类 SortArray 继承类 MyArray, 在该类中定义一个函数, 具有将输入的整数从小到大进行排序的功能。

【提示】

在第(1)步的基础上可增加下面的参考框架:

```

class SortArray : public MyArray {
public:
    void Sort();
    SortArray (int leng):MyArray(leng)
    { cout<<"SortArray 类对象已创建."<<endl;
    }
    virtual ~SortArray ();
};

SortArray::~SortArray ()
{ cout<<"SortArray 类对象被撤销."<<endl;
}

void SortArray::Sort()
{
    //请自行编写 Sort 函数的代码, 将输入的整数从小到大排序。
}

//并将主函数修改为书如下形式
int main()
{ SortArray s(5);
  s.Input(); //输入整数
  s.Display("显示排序以前的"); //显示排序以前的整数
  s.Sort(); //进行排序
  s.Display("显示排序以后的"); //显示排序以后的整数
  return 0;
}

```

(3) 声明一个类 ReArray 继承类 MyArray, 在该类中定义一个函数, 具有将输入的整

数进行倒置的功能。

【提示】

在第(1)步的基础上可增加下面的参考框架：

```
class ReArray: public MyArray {
public:
    void Reverse();
    ReArray(int leng);
    virtual ~ReArray();
};
```

请读者自行编写构造函数、析构函数和倒置函数 ReArray，以及修改主函数。

(4) 声明一个类 AverArray 继承类 MyArray，在该类中定义一个函数，具有求输入的整数平均值的功能。

【提示】

在第(1)步的基础上增加下面的参考框架：

```
class AverArray : public MyArray {
public:
    AverArray (int leng);
    ~AverArray ();
    double Aver();
};
```

请读者自行编写构造函数、析构函数和求平均值函数 Aver(求解整数的平均值)，以及修改主函数。

(5) 声明一个 NewArray 类，同时继承了类 SortArray、ReArray 和 AverArray，使得类 NewArray 的对象同时具有排序、倒置和求平均值的功能。在继承的过程中声明 MyArray 为虚基类，体会虚基类在解决二义性问题中的作用。

习 题

【5.1】有几种继承方式？每种方式的派生类对基类成员的继承性如何？

【5.2】派生类能否直接访问基类的私有成员？若不能，应如何实现？

【5.3】保护成员有哪些特性？保护成员以公有方式或私有方式被继承后的访问特性如何？

【5.4】派生类构造函数和析构函数的调用顺序是怎样的？

【5.5】什么是多重继承？多重继承时，构造函数和析构函数调用顺序是怎样的？

【5.6】使用派生类的主要原因是（ ）。

- | | |
|--------------|--------------|
| A. 提高程序的运行效率 | B. 提高代码的可重用性 |
| C. 加强类的封装性 | D. 实现数据的隐藏 |

【5.7】派生类的对象对基类成员中（ ）是可以访问的。

- | | |
|--------------|--------------|
| A. 公有继承的公有成员 | B. 公有继承的私有成员 |
| C. 公有继承的保护成员 | D. 私有继承的公有成员 |

【5.8】假设已经定义了一个类 student，现在要定义类 derived，它是从 student 私有派生的，定义类 derived 的正确写法是（ ）。

- A. class derived::student private{ ... }; B. class derived::student public { ... };
C. class derived::private student{ ... }; D. class derived::public student { ... };

【5.9】在多继承构造函数定义中，几个基类构造函数用（ ）分隔。

- A. : B. ; C. , D. ::

【5.10】设置虚基类的目的是（ ）。

- A. 简化程序 B. 消除二义性
C. 提高运行效率 D. 减少目标代码

【5.11】若派生类的成员函数不能直接访问基类中继承来的某个成员，则该成员一定是基类中的（ ）。

- A. 私有成员 B. 公有成员
C. 保护成员 D. 保护成员或私有成员

【5.12】类的保护成员，不可以让（ ）来直接访问。

- A. 该类的成员函数 B. 主函数
C. 该类的友元函数 D. 该类的派生类

【5.13】保护继承时，基类的（ ）在派生类中成为保护成员，不能通过派生类的对象来直接访问该成员。

- A. 任何成员 B. 公有成员和保护成员
C. 保护成员和私有成员 D. 私有成员

【5.14】写出下面程序的运行结果。

```
#include<iostream>
using namespace std;
class A{
private:
    int a;
public:
    A()
    { a=0; }
    A(int i)
    { a=i; }
    void Print()
    { cout<<a<<" "; }
};
class B:public A{
private:
    int b1, b2;
public:
    B()
    { b1=0; b2=0; }
    B(int i)
    { b1=i; b2=0; }
    B(int i, int j, int k):A(i), b1(j), b2(k)
    { }
    void Print()
    { A::Print(); }
```

```

        cout<<b1<<"", "<<b2<<endl;
    }
};
int main()
{ B ob1, ob2(1), ob3(3, 6, 9);
  ob1.Print();
  ob2.Print();
  ob3.Print();
  return 0;
}

```

【5.15】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
class Main{
protected:
    char *mainfood;
public:
    Main(char *name)
    { mainfood=name;
    }
};
class Sub{
protected:
    char *subfood;
public:
    Sub(char *name)
    { subfood=name; }
};
class Menu:public Main, public Sub{
public:
    Menu(char *m, char *s):Main(m), Sub(s)
    { }
    void show();
};
void Menu::show()
{ cout<<"主食="<<mainfood<<endl;
  cout<<"副食="<<subfood<<endl;
}
int main()
{ Menu m("bread", "steak");
  m.show();
  return 0;
}

```

【5.16】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
class B1{
    int b1;
public:
    B1(int i)
    { b1=i;

```

```

        cout<<"constructor B1."<<i<<endl;
    }
    void print()
    { cout<<b1<<endl; }
};

class B2{
    int b2;
public:
    B2(int i)
    { b2=i;
      cout<<"constructor B2."<<i<<endl;
    }
    void print()
    { cout<<b2<<endl; }
};

class B3{
    int b3;
public:
    B3(int i)
    { b3=i;
      cout<<"constructor B3."<<i<<endl;
    }
    int getb3()
    { return b3; }
};

class A :public B2, public B1{
    int a; B3 bb;
public:
    A(int i, int j, int k, int l):B1(i), B2(j), bb(k)
    { a=l;
      cout<<"constructor A."<<l<<endl;
    }
    void print()
    { B1::print();
      B2::print();
      cout<<a<<" "<<bb.getb3()<<endl;
    }
};

int main()
{ A aa(1, 2, 3, 4);
  aa.print();
  return 0;
}

```

【5.17】已有如下的类 Time 和 Date，要求设计一个派生类 Birthtime，它继承类 Time 和 Date，并且增加一个数据成员 Childname 用于表示小孩的名字，同时设计主程序显示一个小孩的出生时间和名字。

```

class Time{
public:
    Time(int h, int m, int s)
    { hours=h;
      minutes=m;
      seconds=s;
    }
};

```

```

    }
    void display()
    { cout<<"出生时间:"<<hours<<"时"<<minutes<<"分"<<seconds<<"秒"<<endl;
    }
protected:
    int hours, minutes, seconds;
};
class Date{
public:
    Date(int m, int d, int y)
    { month=m;
      day=d;
      year=y;
    }
    void display()
    { cout<<"出生年月:"<<year<<"年"<<month<<"月"<<day<<"日"<<endl;
    }
protected:
    int month, day, year;
};

```

【5.18】编写一个学生和教师数据输入和显示程序，学生数据有编号、姓名、班号和成绩，教师数据有编号、姓名、职称和部门。要求将编号、姓名输入和显示设计成一个类 `person`，并作为学生数据操作类 `student` 和教师数据操作类 `teacher` 的基类。

【5.19】设计一个虚基类 `base`，包含姓名和年龄私有数据成员以及相关的成员函数；由它派生出领导类 `leader`，包含职务和部门私有数据成员以及相关的成员函数；再由 `base` 派生出工程师类 `engineer`，包含职称和专业私有数据成员以及相关的成员函数；然后由 `leader` 和 `engineer`，派生出主任工程师类 `chairman`。采用相关数据进行测试。

第6章

多态性与虚函数

多态性是面向对象程序设计的重要特征之一。多态性机制不仅增加了面向对象软件系统的灵活性,进一步减少了冗余信息,而且显著提高了软件的可重用性和可扩充性。多态性的应用可以使编程显得更为简捷、便利,它为程序的模块化设计提供了又一手段。

6.1 多态性概述

多态性指向不同的对象发送同一个消息,不同的对象在接收时会产生不同的行为(即方法)。也就是说,每个对象可以用自己的方式去响应共同的消息。消息就是调用函数,不同的行为就是指不同的实现,即执行不同的函数。直观地说,多态性是指用一个名字定义不同的函数,这些函数执行不同但又类似的操作,从而可以使用相同的方式来调用这些具有不同功能的名同函数。这也是人类思维方式的一种模拟。比如一个对象中有很多求面积的行为,显然可以针对不同的图形(例如长方形、三角形、圆等),写出很多不同名称的函数来实现,这些函数的参数个数和类型可以不同。在C++中,可以利用多态性的特征,用相同的函数名来标识这些函数。这样,就可以达到用同样的接口访问不同功能的函数,从而实现“一个接口,多种方法”。

从实现的角度来讲,多态可以划分为两类:编译时的多态性和运行时的多态性。编译时的多态性是通过静态联编实现的,而运行时的多态性则是通过动态联编实现的。

在C++中,编译时多态性主要是通过函数重载和运算符重载实现的。运行时多态性主要是通过虚函数来实现的。函数重载在前面章节中已作了介绍,本章重点介绍虚函数以及由它们提供的多态性,运算符重载将在下一章介绍。

6.2 基类与派生类对象之间的赋值兼容关系

在一定条件下,不同类型的数据之间可以进行类型转换。例如,可以将整型数据赋给双精度型变量。在赋值之前,先把整型数据转换成为双精度型数据,然后再把它赋给双精度型变量。这种不同类型数据之间的自动转换和赋值,称为赋值兼容。在基类和派生类对象之间也有赋值兼容关系,基类和派生类对象之间的赋值兼容规则是指在需要基类对象的任何地方,都可以使用公有派生类的对象来替代。

通过前面的学习我们知道,通过公有继承,派生类保留了基类中除构造函数、析构函数

之外的所有成员，基类的公有或保护成员的访问权限在派生类中全部按原样保留了下来，在派生类外可以调用基类的公有成员函数访问基类的私有成员。因此，公有派生类具有基类的全部功能，凡是基类能够实现的功能，公有派生类都能实现。我们可以将派生类对象的值赋给基类对象，在用到基类对象的时候可以用其子类对象代替。例如，下面声明的两个类：

```
class Base{                                //声明基类 Base
...
};
class Derived:public Base{                 //声明基类 Base 的公有派生类 Derived
...
};
```

根据赋值兼容规则，在基类 Base 的对象可以使用的任何地方，都可以用派生类 Derived 的对象来替代，但只能使用从基类继承来的成员。具体表现在以下几个方面。

(1) 派生类对象可以向基类对象赋值，即用派生类对象中从基类继承来的数据成员，逐个赋值给基类对象的数据成员。例如：

```
Base b;                                //定义基类 Base 的对象 b
Derived d;                             //定义基类 Base 的公有派生类 Derived 的对象 d
b=d;                                   //用派生类 Derived 的对象 d 对基类对象 b 赋值
```

这样赋值的效果是，对象 b 中所有数据成员都将具有对象 d 中对应数据成员的值。

(2) 派生类对象可以初始化基类对象的引用。例如：

```
Base b;                                //定义基类 Base 的对象 b
Derived d;                             //定义基类 Base 的公有派生类 Derived 的对象 d
Base &br=d;                             //定义基类 Base 的对象的引用 br
//并用派生类 Derived 的对象 d 对其初始化
```

(3) 派生类对象的地址可以赋给指向基类对象的指针。例如：

```
Derived d;                             //定义基类 Base 的公有派生类 Derived 的对象 d
Base * bp=&d;                           //把派生类对象的地址&d 赋值给指向基类的指针 bp
//也就是说，指向基类对象的指针 bp 也可以指向派生类对象 d
```

这种形式的转换，是在实际应用程序中最常见到的。

(4) 如果函数的形参是基类对象或基类对象的引用，在调用函数时可以用派生类对象作为实参。例如：

```
class Base{                              //声明基类 Base
public:
    int i;
...
};
class Derived:public Base{               //声明基类 Base 的公有派生类 Derived
...
};
void fun(Base &bb)                       //普通函数，形参为基类 Base 对象的引用
{ cout<<bb.i<<endl;                     //输出该引用所代表的对象的数据成员 i
}
```

在调用函数 fun 时可以用派生类 Derived 的对象 d4 作为实参：

```
fun(d4);
```

输出派生类 Derived 的对象 d4 赋给基类的数据成员 i 的值。

下面是一个使用赋值兼容规则的例子。

例 6.1 基类与派生类对象之间的转换。

```
#include<iostream>
using namespace std;
class Base{                                //声明基类 Base
public:
    int i;
    Base(int x)                            //基类的构造函数
    { i=x; }
    void show()                            //成员函数
    { cout<<"Base "<<i<<endl; }
};
class Derived:public Base{                 //声明公有派生类 Derived
public:
    Derived(int x):Base(x)                 //派生类的构造函数
    { }
};
void fun(Base &bb)                         //普通函数，形参为基类对象的引用
{ cout<<bb.i<<endl; }
int main()
{ Base b1(100);                           //定义基类对象 b1
  b1.show();
  Derived d1(11);                          //定义派生类对象 d1
  b1=d1;                                   //用派生类对象 d1 给基类对象 b1 赋值
  b1.show();
  Derived d2(22);                          //定义派生类对象 d2
  Base &b2=d2;                             //用派生类对象 d2 来初始化基类对象的引用 b2
  b2.show();
  Derived d3(33);                          //定义派生类对象 d3
  Base *b3=&d3;                             //把派生类对象的地址&d3 赋值给指向基类的指针 b3
  b3->show();
  Derived d4(44);                          //定义派生类对象 d4
  fun(d4);                                 //派生类的对象 d4 作为函数 fun 的实参
  return 0;
}
```

程序运行的结果如下：

```
Base 100
```

```
Base 11
```

```
Base 22
```

```
Base 33
```

```
44
```

说明:

(1) 声明为指向基类对象的指针可以指向它的公有派生的对象, 但不允许指向它的私有派生的对象。例如:

```
class Base{
    ...
};
class Derive:private Base
{
    ...
};
int main()
{ Base op1, *ptr;      //定义基类 Base 的对象 op1 及指向基类 Base 的指针 ptr
  Derive op2;          //定义派生类 Derive 的对象 op2
  ptr=&op1;             //将指针 ptr 指向基类对象 op1
  ptr=&op2;             //错误, 不允许将指向基类 Base 的指针 ptr 指向它的私有派生类对象 op2
  ...
}
```

(2) 允许将一个声明为指向基类的指针指向其公有派生类的对象, 但是不能将一个声明为指向派生类对象的指针指向其基类的对象。例如:

```
class Base{
    ...
};
class Derived:public Base{
    ...
};
int main()
{ Base obj1;           //定义基类对象 obj1
  Derived obj2, *ptr;   //定义派生类对象 obj2 及指向派生类的指针 ptr
  ptr=&obj2;            //将指向派生类对象的指针 ptr 指向派生类对象 obj2
  ptr=&obj1;            //错误, 试图将指向派生类对象的指针 ptr 指向其基类对象 obj1
  ...
}
```

6.3 虚 函 数

虚函数是重载的另一种表现形式。这是一种动态的重载方式, 它提供了一种更为灵活的运行时的多态性机制。虚函数允许函数调用与函数体之间的联系在运行时才建立, 也就是在运行时才决定如何动作, 即所谓的动态联编。

6.3.1 虚函数的引入

在引入虚函数的概念之前, 我们先看下面的例题。

例 6.2 虚函数的引例。

```
#include<iostream>
using namespace std;
```

```

class My_base{                                     //声明基类 My_base
public:
    My_base(int x, int y)                          //基类构造函数
    { a=x; b=y;
    }
    void show()                                    //基类的成员函数 show
    { cout<<"调用基类 My_base 的 show() 函数\n";
      cout<<a<<" "<<b<<endl;
    }
private:
    int a, b;
};

class My_class :public My_base{                    //声明派生类 My_class
public:
    My_class(int x, int y, int z):My_base(x, y)    //派生类构造函数
    { c=z; }
    void show()                                    //派生类的成员函数 show
    { cout<<"调用派生类 My_class 的 show() 函数 \n";
      cout<<"c= " <<c<<endl;
    }
private:
    int c;
};

int main()
{ My_base mb(50, 50), *mp;                        //定义基类对象 mb 和对象指针 mp
  My_class mc(10, 20, 30);                        //定义派生类对象 mc
  mp=&mb;                                           //对象指针 mp 指向基类对象 mb
  mp->show();
  mp=&mc;                                           //对象指针 mp 指向派生类对象 mc
  mp->show();
  return 0;
}

```

程序运行结果如下：

```

调用基类 My_base 的 show()函数
50 50
调用基类 My_base 的 show()函数
10 20

```

显然，这个程序的运行结果与我们预想的结果不同，那么为什么会出现这种情况呢？原来，在 C++ 中规定：基类的对象指针可以指向它的公有派生类的对象，但是当其指向公有派生类对象时，它只能访问派生类中从基类继承来的成员，而不能访问公有派生类中定义的成员。因此，从程序运行的结果可以看出，当执行语句

```

mp=&mb;                                           //对象指针 mp 指向基类对象 mb
mp->show();

```

后，指针 mp 指向了基类对象 mb，于是执行语句“mp->show();”调用的基类的函数 show，于是输出：

调用基类 My_base 的 show()函数

50 50

接着, 执行语句

```
mp=&mc;           //对象指针 mp 指向派生类对象 mc
mp->show();
```

后, 指针 mp 指向了派生类对象 mc, 但是执行语句 “mp->show();” 后, 输出的结果却是:

```
调用基类 My_base 的 show() 函数
10 20
```

这说明了: 虽然指针 mp 指向了派生类对象 mc, 但是执行语句 “mp->show();” 调用的不是派生类的成员函数 show, 而仍然是基类的同名成员函数 show, 这显然不是我们所期望的。

运行结果说明, 在这个例子中不管指针 mp 当前指向哪个对象 (基类对象或派生类对象), “mp->show()” 调用的都是基类中定义的 show 函数。

使用对象指针的目的是为了表达一种动态的性质, 即当指针指向不同的对象 (基类对象或派生类对象) 时, 分别调用不同类的成员函数。如果将函数说明为虚函数, 就能实现这种动态调用的功能。

6.3.2 虚函数的作用和定义

1. 虚函数的作用

在例 6.2 中, 虽然基类指针 mp 已经指向了派生类对象 mc, 但是它所调用的成员函数 show, 仍然是其基类对象的成员函数 show。在这种情况下, 若要调用派生类中的成员函数可以采用显式的方法。例如:

```
mc.show();
```

但是, 使用对象指针的目的就是为了表达一种动态的性质, 即当指针指向不同对象时执行不同的操作, 显然以上两种方法都没有起到这种作用。其实只要将成员函数 show 说明为虚函数, 就能实现这种动态调用的功能。

虚函数首先是基类中的成员函数, 但这个成员函数前面缀上关键字 virtual, 并在派生类中被重载。下面的程序将例 6.2 中的函数 show 定义为虚函数, 就能实现动态调用的功能。

例 6.3 虚函数的引入。

```
#include<iostream>
using namespace std;

class My_base{                               //声明基类 My_base
public:
    My_base(int x, int y)                     //基类构造函数
    { a=x; b=y; }
    virtual void show()                       //在基类中定义虚函数 show
    { cout<<"调用基类 My_base 的 show() 函数\n";
      cout<<a<<" "<<b<<endl;
    }
private:
    int a, b;
};
```

```

class My_class :public My_base{           //声明派生类 My_class
public:
    My_class(int x, int y, int z):My_base(x, y) //派生类构造函数
    { c=z;
    }
    virtual void show()                   //在派生类中重新定义虚函数 show
    { cout<<"调用派生类 My_class 的 show() 函数\n";
      cout<<c<<<endl;
    }
private:
    int c;
};

int main()
{ My_base mb(50, 50), *mp;               //定义基类对象 mb 和对象指针 mp
  My_class mc(10, 20, 30);               //定义派生类对象 mc
  mp=&mb;                                 //对象指针 mp 指向基类对象 mb
  mp->show();
  mp=&mc;                                 //对象指针 mp 指向派生类对象 mc
  mp->show();
  return 0;
}

```

程序运行结果如下:

```

调用基类 My_base 的 show()函数
50 50
调用派生类 My_class 的 show()函数
30

```

为什么把基类中的函数 show 定义为虚函数时,程序的运行结果就正确了呢?这是因为,关键字 virtual 指示 C++编译器,函数调用 my->show()要在运行时确定所要调用的函数,即要对该调用进行动态联编。因此,程序在运行时根据指针 mp 所指向的实际对象,调用该对象的成员函数。

我们把使用同一种调用形式“mp->show()”,调用同一类族中不同类的虚函数称为动态的多态性,即运行时的多态性。可见,虚函数可使 C++支持运行时的多态性。

2. 虚函数的定义

虚函数的定义是在基类中进行的,它是在基类中需要定义为虚函数的成员函数的声明中冠以关键字 virtual,从而提供一种接口界面。定义虚函数的方法如下:

```

virtual 返回类型 函数名(形参表)
{
    函数体
}

```

在基类中的某个成员函数被声明为虚函数后,此虚函数就可以在一个或多个派生类中被重新定义。虚函数在派生类中重新定义时,其函数原型,包括返回类型、函数名、参数个数、参数类型的顺序,都必须与基类中的原型完全相同。请看下面的例子。

例 6.4 虚函数的定义举例。

```

#include<iostream>

```

```

using namespace std;
class Grandam{                                //声明基类 Grandam
public:
    virtual void introduce_self()            //定义虚函数 introduce_self
    { cout<<"I am grandam."<<endl; }
};
class Mother:public Grandam{                  //声明派生类 Mother
public:
    void introduce_self()                    //重新定义虚函数 introduce_self
    { cout<<"I am mother."<<endl; }
};
class Daughter:public Mother{                //声明派生类 Daughter
public:
    void introduce_self()                    //重新定义虚函数 introduce_self
    { cout<<"I am daughter."<<endl; }
};
int main()
{
    Grandam *ptr;                             //定义指向基类的对象指针 ptr
    Grandam g;                                //定义基类 Grandam 的对象 g
    Mother m;                                 //定义派生类 Mother 的对象 m
    Daughter d;                              //定义派生类 Daughter 的对象 d
    ptr=&g;                                   //对象指针 ptr 指向基类 Grandam 的对象 g
    ptr->introduce_self();                     //调用基类 Grandam 的虚函数 introduce_self
    ptr=&m;                                   //对象指针 ptr 指向派生类 Mother 的对象 m
    ptr->introduce_self();                     //调用派生类 Mother 的虚函数 introduce_self
    ptr=&d;                                   //对象指针 ptr 指向派生类 Daughter 的对象 d
    ptr->introduce_self();                     //调用派生类 Daughter 的虚函数 introduce_self
    return 0;
}

```

程序只在基类 Grandam 中显式定义了 introduce_self 为虚函数。C++ 规定，如果在派生类中，没有用 virtual 显式地给出虚函数声明，这时系统就会遵循以下的规则来判断一个成员函数是不是虚函数：

- ① 该函数与基类的虚函数有相同的名称；
- ② 该函数与基类的虚函数有相同的参数个数及相同的对应参数类型；
- ③ 该函数与基类的虚函数有相同的返回类型或者满足赋值兼容规则的指针、引用型的返回类型。

派生类的函数满足了上述条件，就被自动确定为虚函数。因此，在本程序的派生类 Mother 中，introduce_self 仍为虚函数，并且在 Mother 的派生类 Daughter 中，introduce_self 还是虚函数。

在主函数 main 中说明了 3 个对象，基类 Grandam 的对象 g，派生类 Mother 的对象 m 和派生类 Daughter 的对象 d。在程序中，语句

```
ptr->introduce_self();
```

出现了 3 次，由于 ptr 指向的对象不同，每次出现都执行了虚函数 introduce_self 的不同版本。

本程序运行的结果如下：

```
I am grandam.
I am mother.
I am daughter.
```

下面对虚函数的定义作几点说明：

(1) 由于虚函数使用的基础是赋值兼容规则，而赋值兼容规则成立的前提条件是派生类从其基类公有派生。因此，通过定义虚函数来使用多态性机制时，派生类必须从它的基类公有派生。

(2) 必须首先在基类中定义虚函数。由于“基类”与“派生类”是相对的，因此，这项说明并不表明必须在类等级的最高层类中声明虚函数。在实际应用中，应该在类等级内需要具有动态多态性的几个层次中的最高层类内首先声明虚函数。

(3) 在派生类对基类中声明的虚函数进行重新定义时，关键字 `virtual` 可以写也可以不写。但在容易引起混乱的情况下，最好在对派生类的虚函数进行重新定义时也加上关键字 `virtual`。

(4) 虽然使用对象名和点运算符的方式也可以调用虚函数，例如语句

```
m.introduce_self();
```

可以调用虚函数 `Mother::introduce_self()`。但是这种调用在编译时进行的是静态联编，它没有充分利用虚函数的特性。只有通过基类指针访问虚函数时才能获得运行时的多态性。

(5) 一个虚函数无论被公有继承多少次，它仍然保持其虚函数的特性。

(6) 虚函数必须是其所所在类的成员函数，而不能是友元函数，也不能是静态成员函数，因为虚函数调用要靠特定的对象来决定该激活哪个函数。

(7) 内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时仍将其看作是非内联的。

(8) 构造函数不能是虚函数，但是析构函数可以是虚函数，而且通常说明为虚函数。

3. 虚函数的简单应用举例

例 6.5 应用 C++ 的多态性，计算三角形、矩形和圆的面积。

```
#include<iostream>
using namespace std;
class Figure{                               //定义一个公共基类
public:
    Figure(double a, double b)
    { x=a; y=b;}
    virtual void area()                     //定义一个虚函数，作为界面接口
    { cout<<"在基类中定义的虚函数，";
      cout<<"为派生类提供一个公共接口，";
      cout<<"以便派生类根据需要重新定义虚函数."<<endl;
    }
protected:
    double x, y;
};
class Triangle:public Figure{               //定义三角形派生类
public:
    Triangle(double a, double b):Figure(a, b)
```

```

    { };
    void area() //虚函数重新定义,用作求三角形的面积
    { cout<<"三角形的高是"<<x<<" , 底是 "<<y;
      cout<<" , 面积是"<<0.5*x*y<<endl;
    }
};

class Square:public Figure{ //定义矩形派生类
public:
    Square(double a, double b):Figure(a, b)
    { };
    void area() //虚函数重新定义,用作求矩形的面积
    { cout<<"矩形的长是"<<x<<" , 宽是 "<<y;
      cout<<" , 面积是"<<x*y<<endl;
    }
};

class Circle:public Figure{ //定义圆派生类
public:
    Circle(double a):Figure(a, a)
    { };
    void area() //虚函数重新定义,用作求圆的面积
    { cout<<"圆的半径是"<<x;
      cout<<" , 面积是"<<3.1416*x*x<<endl;
    }
};

int main()
{ Figure *p; //定义基类指针 p
  Triangle t(10.0, 6.0); //定义三角形类对象 t
  Square s(10.0, 6.0); //定义矩形类对象 s
  Circle c(10.0); //定义圆类对象 c
  p=&t;
  p->area(); //计算三角形面积
  p=&s;
  p->area(); //计算矩形面积
  p=&c;
  p->area(); //计算圆面积
  return 0;
}

```

程序运行结果如下:

三角形的高是 10, 底是 6, 面积是 30

矩形的长是 10, 宽是 6, 面积是 60

圆的半径是 10, 面积是 314.16

分析以上程序可知,由于在公共基类 Figure 中定义一个虚函数 area 作为界面接口,在 3 个派生类 Triangle、Square 和 Circle 中重新定义了虚函数 area,分别用于计算三角形、矩形和圆形的面积。由于 p 是基类的对象指针,用同一种调用形式“p->area()”,就可以调用同一类族中不同类的虚函数。这就是多态性,对于同一条消息,不同的对象有不同的响应方式。

6.3.3 虚析构函数

在第5章曾经介绍,当派生类对象撤销时,一般先调用派生类的析构函数,然后再调用基类的析构函数。请看下面的例子。

例 6.6 虚析构函数的引例 1。

```
#include<iostream>
using namespace std;
class Base{
public:
    ~Base()
    { cout<<"调用基类 Base 的析构函数\n"; }
};
class Derived:public Base{
public:
    ~Derived()
    { cout<<"调用派生类 Derived 的析构函数\n"; }
};
int main()
{ Derived obj;
  return 0;
}
```

程序运行的结果如下:

```
调用派生类 Derived 的析构函数
调用基类 Base 的析构函数
```

显然本程序的运行结果是符合预想结果的。但是,如果在主函数中用 `new` 运算符建立一个派生类的无名对象和定义了一个基类的对象指针,并将无名对象的地址赋给这个对象指针。当用 `delete` 运算符撤销无名对象时,系统只执行基类的析构函数,而不执行派生类的析构函数。

例 6.7 虚析构函数的引例 2。

```
#include<iostream>
using namespace std;
class Base{
public:
    ~Base()
    { cout<<"调用基类 Base 的析构函数\n"; }
};
class Derived:public Base{
public:
    ~Derived()
    { cout<<"调用派生类 Derived 的析构函数\n"; }
};
int main()
{ Base *p; //定义指向基类 Base 的指针变量 p
  p= new Derived; //用运算符 new 为派生类的无名对象动态地分配了一个存储空间
                  //并将地址赋给对象指针 p
  delete p; //用 delete 撤销无名对象,释放动态存储空间
  return 0;
}
```

程序运行的结果如下：

调用基类 Base 的析构函数

运行结果表明，本程序只执行了基类 Base 的析构函数，而没有执行派生类 Derived 的析构函数。原因是当撤销指针 P 所指的派生类的无名对象，而调用析构函数时，采用了静态联编方式，只调用了基类 Base 的析构函数。

如果希望程序执行动态联编的方式，在用 delete 运算符撤销派生类的无名对象时，先调用派生类的析构函数，再调用基类的析构函数，可以将基类的析构函数声明为虚析构函数。虚析构函数没有类型，也没有参数，虚析构函数的定义比较简单。其定义的一般格式为：

```
virtual ~类名()
{
    函数体
};
```

虽然派生类的析构函数与基类的析构函数名字不相同，但是如果将基类的析构函数定义为虚函数，由该基类派生的所有派生类的析构函数也都自动成为虚函数。请看下面的例子。

例 6.8 虚析构函数的使用。

```
#include<iostream>
using namespace std;
class Base{
public:
    virtual ~Base()           //将基类的析构函数声明为虚析构函数
    { cout<<"调用基类 Base 的析构函数\n"; }
};
class Derived:public Base{
public:
    ~Derived()                //派生类的析构函数也都自动成为虚函数
    { cout<<"调用派生类 Derived 的析构函数\n"; }
};
int main()
{ Base *p;                   //定义指向基类 Base 的指针变量 p
  p= new Derived;             //用运算符 new 为派生类的无名对象动态地分配了一个存储空间
                               //并将地址赋给对象指针 p
  delete p;                   //用 delete 撤销无名对象，释放动态存储空间
  return 0;
}
```

在这个程序中，将例 6.5 中基类的析构函数声明为虚析构函数，程序的其他部分没有改动。但是运行程序后，结果变为：

调用派生类 Derived 的析构函数

调用基类 Base 的析构函数

显然，这个结果是符合人们的愿望的。这是由于使用了虚析构函数，程序执行了动态联编，实现了运行的多态性。

6.4 纯虚函数和抽象类

6.4.1 纯虚函数

有时，基类往往表示一种抽象的概念，它并不与具体的事物相联系。这时在基类中将某一成员函数定义为虚函数，并不是基类本身的要求，而是考虑到派生类的需要，在基类中预留了一个函数名，具体功能留给派生类根据需要去定义。如例 6.9 中，Shape 是一个基类，它表示具有封闭图形的东西。从 Shape 可以派生出三角形类、矩形类和圆类。在这个类等级中基类 Shape 体现了一个抽象的概念，在基类 Shape 中定义一个求面积的函数显然是无意义的，但是我们可以将其说明为虚函数，为它的派生类提供一个公共的界面，各派生类根据所表示的图形的不同，重定义这些虚函数，以提供求面积的各自版本。为此，C++ 列入了纯虚函数的概念。

纯虚函数是在声明虚函数时被“初始化”为 0 的函数。声明纯虚函数的一般形式如下：
virtual 函数类型 函数名(参数表)=0;

此格式与一般的虚函数定义格式基本相同，只是在后面多了“=0”。声明为纯虚函数之后，基类中就不再给出函数的实现部分。假如在例 6.9 中，将基类 Shape 中虚函数 show_area 写成纯虚函数，格式如下：

```
virtual void show_area()=0;
```

纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行重新定义。纯虚函数没有函数体，它最后面的“=0”并不表示函数的返回值为 0，它只起形式上的作用，告诉编译系统“这是纯虚函数”。纯虚函数不具备函数的功能，不能被调用。

例 6.9 应用 C++ 的多态性，计算三角形、矩形和圆的面积。

```
#include<iostream>
using namespace std;
class Shape{                                //定义一个公共基类
public:
    Shape(double a, double b)
    { x=a; y=b; }
    virtual void show_area()=0;              //定义一个纯虚函数，作为界面接口
    protected:
        double x, y;
};
class Triangle:public Shape{                 //定义三角形派生类
public:
    Triangle(double a, double b):Shape(a, b)
    { };
    void show_area()                          //虚函数重定义，用作求三角形的面积
    { cout<<"Triangle with height "<<x;
      cout<<" and base "<<y<<" has an area of ";
      cout<<x*y*0.5<<endl;
    }
};
```

```

};
class Square:public Shape{           //定义矩形派生类
public:
    Square(double a, double b):Shape(a, b)
    { };
    void show_area()                 //虚函数重载定义, 用作求矩形的面积
    { cout<<"Square with dimension "<<x;
      cout<<" * "<<y<<" has an area of ";
      cout<<x*y<<endl;
    }
};

class Circle:public Shape{          //定义圆派生类
public:
    Circle(double a):Shape(a, a)
    { };
    void show_area()                 //虚函数重载定义, 用作求圆的面积
    { cout<<"Circle with radius "<<x;
      cout<<" has an area of ";
      cout<<x*x*3.1416<<endl;
    }
};

int main()
{ Shape *p;                          //定义基类指针 p
  Triangle t(10.0, 6.0);              //定义三角形类对象 t
  Square s(10.0, 6.0);                //定义矩形类对象 s
  Circle c(10.0);                     //定义圆类对象 c
  p=&t;
  p->show_area();                      //计算三角形面积
  p=&s;
  p->show_area();                      //计算矩形面积
  p=&c;
  p->show_area();                      //计算圆面积
  return 0;
}

```

程序运行结果如下:

Triangle with height 10 and base 6 has an area of 30

Square with dimension 10 * 6 has an area of 60

Circle with radius 10 has an area of 314.16

通过分析以上程序可知, 由于在公共基类 Shape 中定义一个虚函数 show_area 作为界面接口, 在 3 个派生类 Triangle、Square 和 Circle 中重新定义了虚函数 show_area, 分别用于计算三角形、矩形和圆形的面积。由于 p 是基类的对象指针, 用同一种调用形式 “p->show_area();”, 就可以调用同一类族中不同类的虚函数。这就是多态性, 对同一条消息, 不同的对象有不同的响应方式。

6.4.2 抽象类

如果一个类至少有一个纯虚函数, 那么就称该类为抽象类。因此, 上述程序中定义的一类 Shape 就是一个抽象类。定义抽象类的唯一目的是用它作为基类去建立派生类。抽象类作为

一种基本类型提供给用户，用户在这个基础上根据自己的需要定义出功能各异的派生类，并用这些派生类去建立对象。对于抽象类的使用有以下几点规定。

(1) 由于抽象类中至少包含一个没有定义功能的纯虚函数。因此，抽象类只能作为其他类的基类来使用，不能建立抽象类对象。

(2) 不允许从具体类派生出抽象类。所谓具体类，就是不包含纯虚函数的普通类。

(3) 抽象类不能用作函数的参数类型、函数的返回类型或显式转换的类型。

(4) 可以声明指向抽象类的指针或引用，此指针可以指向它的派生类，进而实现多态性。

(5) 如果派生类中没有定义纯虚函数的实现，而派生类只是继承基类的纯虚函数，则这个派生类仍然是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体类了。

6.5 应用举例

例 6.10 应用抽象类，求圆、圆内接正方形和圆外切正方形的面积和周长。程序如下：

```
#include<iostream>
using namespace std;
class Shape{                                //声明一个抽象类
public:
    Shape(double x)
    { r=x; }
    virtual void area()=0;                  //纯虚函数
    virtual void perimeter()=0;            //纯虚函数
protected:
    double r;
};
class Circle:public Shape{                  //声明一个圆派生类
public:
    Circle(double x):Shape(x)
    { }
    void area()                             //定义虚函数 area
    { cout<<"这个圆的面积是: "<<3.14*r*r<<endl; }
    void perimeter()                        //定义虚函数 perimeter
    { cout<<"这个圆的周长是: "<<2*3.14*r<<endl; }
};
class In_square:public Shape{               //声明一个圆内接正方形类
public:
    In_square(double x):Shape(x)
    { }
    void area()                             //定义虚函数 area
    { cout<<"这个圆内接正方形的面积是: "<<2*r*r<<endl; }
    void perimeter()                        //定义虚函数 perimeter
    { cout<<"这个圆内接正方形的周长是: "<<4*1.414*r<<endl; }
};
class Ex_square:public Shape               //声明一个圆外切正方形类
```

```

{ public:
    Ex_square(double x):Shape(x)
    { }
    void area()                //定义虚函数 area
    { cout<<"这个圆外切正方形的面积是:"<<4*r<<endl; }
    void perimeter()           //定义虚函数 perimeter
    { cout<<"这个圆外切正方形的周长是:"<<8*r<<endl; }
};

int main()
{ Shape *ptr;                 //定义抽象类 Shape 的对象指针 ptr
    Circle ob1(5);             //定义圆类 Circle 的对象 ob1
    In_square ob2(5);          //定义圆内接正方形类 In_square 的对象 ob2
    Ex_square ob3(5);          //定义圆外切正方形类 Ex_square 的对象 ob3
    ptr=&ob1;                   //指针 ptr 指向 Circle 类对象 ob1
    ptr->area();                 //求圆的面积
    ptr->perimeter();            //求圆的周长
    ptr=&ob2;                   //指针 ptr 指向 In_square 类对象 ob2
    ptr->area();                 //求圆内接正方形的面积
    ptr->perimeter();            //求圆内接正方形的周长
    ptr=&ob3;                   //指针 ptr 指向 Ex_square 类对象 ob3
    ptr->area();                 //求圆外切正方形的面积
    ptr->perimeter();            //求圆外切正方形的周长
    return 0;
}

```

程序运行结果如下:

```

这个圆的面积是: 78.5
这个圆的周长是: 31.4
这个圆内接正方形的面积是: 50
这个圆内接正方形的周长是: 28.28
这个圆外切正方形的面积是: 100
这个圆外切正方形的周长是: 40

```

在以上程序中, 声明公共基类 Shape 为抽象类, 在其中定义求面积和周长的纯虚函数 area 和 perimeter 作为界面对接口。抽象类 Shape 有 3 个派生类 Circle、In_square 和 Ex_square, 分别求圆、圆内接正方形和圆外切正方形的面积和周长。根据各自的功能, 每个派生类定义了虚函数 area 和 perimeter, 以计算出各自图形的面积和周长。我们可以看到, 尽管在 3 个派生类 Circle、In_square 和 Ex_square 中对虚函数 area 和 perimeter 定义的功能各不相同, 但接口都是抽象基类 Shape 中的纯虚函数 area 和 perimeter。

抽象类和虚函数使程序的扩充非常容易。例如, 在上述程序中, 可以通过在 main 函数前增加下述派生类的定义, 即可增加一个计算圆外切三角形面积和周长的功能:

```

class Triangle:public Shape{
public:
    Triangle(double x):Shape(x)
    { }
    void area()

```



```

{ cout<<"这个圆外切三角形的面积是:"<<3*1.732*r<<endl; }
void perimeter()
{ cout<<"这个圆外切三角形的周长是:"<< 6*1.732*r<<endl; }
};

```

如果在 main 函数中增加下述几条语句:

```

Trinagle ob4(5);           //定义圆外切三角形类 Trinagle 的对象 ob4
ptr=&ob4;                  //指针 ptr 指向 Trinagle 类对象 ob4
ptr->area();                //求圆外切三角形的面积
ptr->perimeter();           //求圆外切三角形的周长

```

程序运行后,即可打印出相应三角形的面积和周长。

这个圆外切三角形的面积是: 129.9

这个圆外切三角形的周长是: 51.96

实 验

实验目的和要求

1. 了解多态性的概念。
2. 学习虚函数的定义和使用方法。
3. 学习纯虚函数和抽象类的概念和用法。

实验内容和步骤

1. 分析并调试下列程序,写出程序的输出结果,并分析输出结果。

```

//test6_lcpp
#include<iostream>
using namespace std;
class Stock{
public:
    void print()
    { cout<<"Stock class.\n"; }
};
class Der1_Stock:public Stock{
public:
    void print()
    { cout<<"Der1_Stock class.\n"; }
};
class Der2_Stock: public Stock{
public:
    void print()
    { cout<<"Der2_Stock class.\n"; }
};
int main()
{ Stock s1;
  Stock *ptr;
  Der1_Stock d1;
  Der2_Stock d2;
  ptr=&s1;
  ptr->print();
  ptr=&d1;

```

```
ptr->print();
ptr=&d2;
ptr->print();
return 0;
}
```

2. 修改上一题的程序, 使运行结果为:

```
Stock class.
Der1_Stock class.
Der2_Stock class.
```

3. 声明抽象基类, 由它派生出 3 个派生类: Triangle (等腰三角形)、Square (矩形) 和 Circle (圆形)。应用 C++ 的多态性, 计算并显示等腰三角形、矩形和圆形的周长。

4. 给出下面的抽象基类 Container:

```
class Container{                                //声明抽象类 Container
protected:
    double radius;
public:
    Container(double radius1);                //抽象类 Container 的构造函数
    virtual double surface_area()=0;          //纯虚函数 surface_area
    virtual double volume()=0;                //纯虚函数 volume
};
```

要求建立 3 个继承 Container 的派生类 cube、sphere 与 cylinder, 让每一个派生类都包含虚函数 surface_area() 和 volume(), 分别用来计算正方体、球体和圆柱体的表面积及体积。要求写出主程序, 应用 C++ 的多态性, 分别计算边长为 6.0 的正方体、半径为 5.0 的球体, 以及半径为 5.0 和高为 6.0 的圆柱体的表面积和体积。

习 题

【6.1】从实现的角度来讲, 多态可以划分为哪两类? 在 C++ 中, 它们主要是通过什么方法实现的?

【6.2】实现编译时多态性主要是通过 () 实现的。

A. 重载函数 B. 友元函数 C. 构造函数 D. 虚函数

【6.3】实现运行时的多态性主要是通过 () 实现的。

A. 重载函数 B. 友元函数 C. 构造函数 D. 虚函数

【6.4】关于虚函数, 正确的描述是 ()。

A. 构造函数不能是虚函数 B. 析构函数不能是虚函数
C. 虚函数可以是友元函数 D. 虚函数可以是静态成员函数

【6.5】要实现动态联编, 派生类中的虚函数 ()。

A. 返回的类型可以与虚函数的原型不同 B. 参数个数可以与虚函数的原型不同
C. 参数类型可以与虚函数的原型不同 D. 以上都不对

【6.6】虚函数是用关键字 () 标记的。

A. virtual B. static C. public D. inline

【6.7】如果在基类中将函数 show 声明为不带返回值的纯虚函数，正确的写法是（ ）。

- A. virtual show()=0; B. virtual void show();
C. virtual void show()=0; D. void show()=0 virtual;

【6.8】如果一个类至少有一个纯虚函数，那么该类称为（ ）。

- A. 抽象类 B. 虚基类 C. 派生类 D. 以上都不对

【6.9】下列关于纯虚函数与抽象类的描述中，错误的是（ ）。

- A. 纯虚函数是一种特殊的函数，它允许没有具体的实现
B. 抽象类是指具有纯虚函数的类
C. 一个基类的说明中有纯虚函数，该基类的派生类一定不再是抽象类
D. 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出

【6.10】有如下程序：

```
#include<iostream>
using namespace std;
class shapes
{ protected:
    int x, y;
public:
    void setvalue(int d, int w=0)
    { x=d; y=w; }
    virtual void disp()=0;
};

class square:public shapes{
public:
    void disp()
    { cout<<"x*y<<endl; }
};

int main()
{ shapes *ptr;
  square s1;
  ptr=&s1;
  ptr->setvalue(10, 5);
  ptr->disp();
  return 0;
}
```

执行上面的程序将输出 ()。

- A. 50 B. 5 C. 10 D. 15

【6.11】下面的程序段中虚函数被重新定义的方法正确吗？为什么？

```
class base {
public:
    virtual int f(int a)=0;
    ...
};
class derived:public base{
public:
    int f(int a, int b)
    { return a*b; }
    ...
}
```

};

【6.12】给出下面的基类：

```
class area_cl {
protected:
    double height;
    double width;
public:
    area_cl(double r, double s)
    { height=r; width=s; }
    virtual double area()=0;
};
```

要求：

(1) 建立基类 `area_cl` 的两个派生类 `rectangle` 与 `isosceles`，让每一个派生类都包含一个函数 `area()`，分别用来返回矩形与三角形的面积。用构造函数对 `height` 与 `width` 进行初始化。

(2) 写出主程序，用来求 `height` 与 `width` 分别为 10.0 与 5.0 的矩形面积，以及求 `height` 与 `width` 分别为 4.0 与 6.0 的三角形面积。

(3) 要求通过使用基类指针访问虚函数的方法（即运行时的多态性）分别求出矩形和三角形面积。

【6.13】定义基类 `Base`，其数据成员为高 `h`，定义成员函数 `disp` 为虚函数。然后再由 `High` 派生出长方体类 `Cuboid` 与圆柱体类 `Cylinder`。并在两个派生类中定义成员函数 `disp` 为虚函数。在主函数中，用基类 `Base` 定义指针变量 `pc`，然后用指针 `pc` 动态调用基类与派生类中虚函数 `disp`，显示长方体与圆柱体的体积。

【6.14】给出下面的抽象基类 `container`：

```
class container{                                //声明抽象类 container
protected:
    double radius;
public:
    container(double radius1);                  //抽象类 container 的构造函数
    virtual double surface_area()=0;            //纯虚函数 surface_area
    virtual double volume()=0;                  //纯虚函数 volume
};
```

要求建立 3 个继承 `container` 的派生类 `cube`、`sphere` 与 `cylinder`，让每一个派生类都包含虚函数 `surface_area()` 和 `volume()`，分别用来计算正方体、球体和圆柱体的表面积及体积。要求写出主程序，应用 C++ 的多态性，分别计算边长为 6.0 的正方体、半径为 5.0 的球体，以及半径为 5.0 和高为 6.0 的圆柱体的表面积和体积。

第 7 章

运算符重载

运算符重载是面向对象程序设计的重要特征。运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据导致不同的行为。在 C++ 中经重载后的运算符能直接对用户自定义的数据进行操作运算。本章将重点介绍有关运算符重载方面的内容。为了能自由地使用重载后的运算符，往往需要在自定义的数据类型和预定义的数据类型之间进行相互转换，或者需要在不同的自定义数据类型之间进行相互转换，因此，本章还将讲述类类型的转换。

7.1 运算符重载概述

运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据导致不同的行为。为什么要重载运算符？如何进行运算符重载？运算符重载能带来哪些好处呢？让我们先看下面的程序片断：

```
int sum_i;
int i1=123, i2=456;
sum_i = i1+i2;
cout<<"i1+i2 ="<< sum_i;
float sum_f;
float f1=3.45, f2=56.78;
sum_f = f1+f2;
cout<<"f1+f2 ="<< sum_f;
```

从上面的程序片断可以看到，在表达式 $i1+i2$ 中的加号“+”用于完成两个整型数的加运算，在表达式 $f1+f2$ 中的加号“+”用于完成两个浮点数的加运算。为什么同一个运算符“+”可以用于完成不同类型的数据的加运算呢？原来 C++ 针对预定义基本数据类型已经对“+”运算符做了适当的重载。

当编译程序编译表达式 $i1+i2$ 的时候，自动使用整型数相加的算法；编译表达式 $f1+f2$ 时，编译程序自动使用浮点数相加的算法。上述这些工作都是编译程序自动完成的，无需程序员操心。有了针对预定义的基本数据类型的运算符重载，给我们编程带来很多方便，而且写出的表达式与数学表达式很相似，符合人们的习惯。试想，如果对于不同类型的数据做同样运算（例如，加法）时，需要使用不同的运算符，那将使人感到多么麻烦，多么不习惯！

但是，C++ 提供的预定义的基本数据类型终究是有限的，我们在解决多种多样的实际问

题时，往往需要使用许多的自定义数据类型。例如，在解决科学与工程计算问题时，往往需要使用复数、矩阵等。

下面我们定义一个简化的复数类 Complex:

```
class Complex{
public:
    double real, imag;
    Complex(double r=0, double i=0)
    { real=r; imag=i;}
};
```

若要把类 Complex 的两个对象 com1 和 com2 直接加在一起，下面的语句是不能实现的：

```
int main()
{ Complex com1(1.1, 2.2), com2(3.3, 4.4), total;
  total=com1+com2;      //错误
  ...
  return 0;
}
```

不能实现的原因是 Complex 类类型不是预定义的基本数据类型，而是用户自定义的数据类型。C++知道如何相加两个 int 型数据，或相加两个 float 型数据，甚至知道如何把一个 int 型数据与一个 float 型数据相加，但是 C++还无法直接将两个 Complex 类对象相加。

为了表达上的方便，人们希望预定义的运算符（如 +、-、*、/等）在特定类的对象上以新的含义进行解释，如希望能够实现 total=com1+com2，这就需要通过重载运算符“+”来解决。

C++为运算符重载提供了一种方法，即在进行运算符重载时，必须写一个运算符函数，其名字规定为 operator 后随一个要重载的运算符。例如，要重载“+”号，应该写一个名字为 operator+ 的函数。其他重载运算符也应该以同样的方式命名，如表 7.1 所示。

表 7.1 运算符重载函数

函 数	功 能
operator+()	加法
operator-()	减法
operator*()	乘法
operator/()	除法
operator<()	小于
...	...

这样，在编译时遇到名为 operator@ 的运算符函数(@表示所要重载的运算符)，就检查传递函数的参数类型。如果编译器在一个运算符的两边“看”到自定义的数据类型，就执行用户自己的函数，而不是这些运算符的常规程序。

因此，若要将上述类 Complex 的两个对象相加，只要编写一个运算符函数 operator+，如下所示：

```
Complex operator+(Complex om1, Complex om2)
{ Complex temp;
  temp.real=om1.real+om2.real;
```

```
temp.imag=com1.imag+com2.imag;
return temp;
}
```

我们就能方便地使用语句：

```
total=com1+com2;
```

将类 `Complex` 的两个对象 `com1` 和 `com2` 相加。当然，在程序中也可以使用以下的调用语句，将两个 `Complex` 类对象相加：

```
total=operator+(com1, com2);
```

这两个调用语句是等价的，但显然后者不如前者直观和方便。

以下就是使用运算符重载函数 `operator+()` 将两个 `Complex` 类对象相加的完整程序。

例 7.1 将两个 `Complex` 类对象相加。

```
#include<iostream>
using namespace std;
class Complex{                                //声明复数类 Complex
public:
    double real;                               //复数实部
    double imag;                               //复数虚部
    Complex(double r=0, double i=0) {
        real=r; imag=i;
    }
};
Complex operator+(Complex col, Complex co2) //定义复数相加的函数
{
    Complex temp;
    temp.real=col.real+co2.real;
    temp.imag=col.imag+co2.imag;
    return temp;
}
int main()
{
    Complex com1(1.1, 2.2), com2(3.3, 4.4), total1, total2;
    total1=operator+(com1, com2);              //调用运算符重载函数 operator+ 的第 1 种方式
    cout<<"real1="<<total1.real<<" "<<"imag1="<<total1.imag<<endl;
    total2=com1+com2;                          //调用运算符重载函数 operator+ 的第 2 种方式
    cout<<"real2="<<total2.real<<" "<<"imag2="<<total2.imag<<endl;
    return 0;
}
```

程序运行结果如下：

```
real1=4.4 imag1=6.6
real2=4.4 imag2=6.6
```

在本例中，`Complex` 的类对象分别使用了两种不同的方式相加，显然使用第 2 种方法，即使用一个简单的“+”号将两个类对象相加更方便明了。实际上，C++编译系统也是将程序中的语句“`total1=com1+com2;`”解释为

```
total1=operator+(com1, com2);
```

来进行处理的。

从本例可以看出，针对类 `Complex` 重载了运算符“+”之后，复数加法的书写形式变得

十分简单(当多个复数对象相加时,书写简单的优点更加明显),并且和预定义类型数据加法的书写形式一样符合人的习惯。

总之,运算符重载进一步提高了面向对象软件系统的灵活性、可扩充性和可读性。

7.2 运算符重载函数作为类的友元函数和成员函数

运算符重载是通过创建运算符重载函数来实现的,运算符重载函数定义了重载的运算符将要进行的操作。例 7.1 中的运算符重载函数是在类的外部定义的,这个运算符重载函数只能访问类中的公有数据成员,而不能访问类的私有数据成员。实际上,运算符重载函数一般采用如下两种形式:一是定义为它将要操作的类的成员函数;二是定义为类的友元函数。

7.2.1 运算符重载函数作为类的友元函数

在 C++ 中,可以把运算符重载函数作为类的友元函数。

1. 运算符重载函数作为类的友元函数的语法形式

运算符函数重载为类的友元函数,则在声明类 X 时,在类的内部函数原型的声明格式如下:

```
class X {
    ...
    friend 返回类型 operator 运算符(形参表);
    ...
}
```

在类外定义友元运算符函数的格式如下:

```
返回类型 operator 运算符(形参表)
{
    函数体
}
```

其中, X 是重载此运算符的类名,返回类型指定了运算符函数的返回值类型; operator 是定义运算符函数的关键字;运算符即是要重载的运算符名称,必须是 C++ 中可重载的运算符;形参表中给出重载运算符所需要的参数和类型;关键字 friend 表明这是一个友元函数。由于友元函数不是该类的成员函数,所以在类外定义时不需要缀上“类名::”。

友元函数没有 this 指针,若友元函数重载的是双目运算符,则参数表中有两个操作数;若重载的是单目运算符,则参数表中只有一个操作数。下面分别予以介绍。

2. 用友元函数重载双目运算符

双目运算符(或称二元运算符)有两个操作数,通常在运算符的左右两侧,例如 $3+5$, $24>12$ 等。当用友元函数重载双目运算符时,两个操作数都要传递给运算符重载函数。下面是一个用友元运算符重载函数进行复数运算的例子。

例 7.2 将运算符函数重载为友元函数，用于进行复数的加法运算。

两个复数 $a+bi$ 和 $c+di$ 进行加法运算的方法如下：

$$(a+bi) + (c+di) = (a+c) + (b+d)i$$

在本例中，声明了一个复数类 `Complex`，类中含有两个数据成员，即复数的实数部分 `real` 和复数的虚数部分 `imag`。下面是这个例子的完整程序。

```
#include<iostream>
using namespace std;

class Complex{                                //声明复数类 Complex
public:
    Complex(double r=0.0, double i=0.0);      //声明用友元函数重载运算符 "+"
    friend Complex operator +(Complex& a, Complex& b);
    void display();
private:
    double real;                               //复数实部
    double imag;                               //复数虚部
};

Complex::Complex(double r, double i)          //构造函数
{ real=r; imag=i; }

Complex operator +(Complex& a, Complex& b)    //重载运算符 "+" 的实现
{ Complex temp;
  temp.real=a.real+b.real;
  temp.imag=a.imag+b.imag;
  return temp;
}

void Complex::display()                       //显示输出复数
{ cout<<real;
  if (imag>0) cout<<"+";
  if (imag!=0) cout<<imag<<"i\n";
}

int main()
{ Complex com1(2.3, 4.6), com2(3.6, 2.8), com3; //定义 3 个复数类对象
  com1.display();                             //输出复数 com1
  com2.display();                             //输出复数 com2
  com3=com1+com2;                             //复数相加
  com3.display();                             //输出复数相加结果 com3
  return 0;
}
```

程序运行结果如下：

```
2.3+4.6i
3.6+2.8i
5.9+7.4i
```

在主函数 `main` 中的语句：

```
com3=com1+com2;
```

C++将其解释为：

```
com3=operator +(com1, com2);
```

一般而言,如果在类 X 中采用友元函数重载双目运算符@,而 aa 和 bb 是类 X 的两个对象,则以下两种函数调用方法是等价的:

```
aa @ bb;           //隐式调用
operator @(aa, bb); //显式调用
```

说明:

(1) 在函数返回的时候,有时可以直接用类的构造函数来生成一个临时对象,而不对该对象进行命名,如可将上例重载运算符“+”的友元函数

```
Complex operator+(Complex& a, Complex& b)
{ Complex temp;
  temp.real=a.real+b.real;
  temp.imag=a.imag+b.imag;
  return temp;
}
```

改写为

```
Complex operator +(Complex& a, Complex& b)
{ return Complex(a.real+b.real, a.imag+b.imag);
}
```

其中 return 语句中的

```
Complex(a.real+b.real, a.imag+b.imag);
```

是建立一个临时对象,它没有对象名,是一个无名对象。在建立临时对象过程中调用构造函数,return 语句将此临时对象作为函数返回值。这种方法执行的效率比较高,但前一种方法可读性比较好。

(2) 有的 C++ 系统(如 Visual C++6.0)没有完全实现 C++ 标准,它所提供的不带后缀的.h 的头文件不支持把运算符函数重载为友元函数,在 Visual C++ 6.0 中编译会出错,这时可采用带后缀的.h 头文件。将程序中的

```
#include<iostream>
using namespace std;
```

修改成

```
#include<iostream.h>
```

即可顺利运行。以后遇到类似情况,可照此办理。

3. 用友元函数重载单目运算符

单目运算符只有一个操作数,如 -a, &b, !c, ++p 等。重载单目运算符的方法与重载双目运算符的方法是类似的。用友元函数重载单目运算符时,需要一个显式的操作数。下面的例子,用友元函数重载单目运算符“++”。

例 7.3 用友元函数重载单目运算符“++”。

```
#include<iostream>
using namespace std;
class Coord{                               //声明类 Coord
public:
  Coord(int i=0, int j=0);
```

```

void display();
friend Coord operator ++(Coord &op);           //声明用友元函数重载运算符“++”
private:                                       //采用引用参数传递操作数
    int x, y;
};
Coord::Coord(int i, int j)
{ x=i; y=j; }
void Coord::display()
{ cout<<"  x: "<<x<<" , y: "<<y<<endl; }
Coord operator ++(Coord &op)                 //重载运算符“++”的实现
{ ++op.x;
  ++op.y;
  return op;
}
int main()
{ Coord ob(11, 22);
  ob.display();
  ++ob;                                       //隐式调用运算符函数 operator++
  ob.display();
  operator++(ob);                             //显式调用运算符函数 operator++
  ob.display();
  return 0;
}

```

程序运行的结果如下:

```

x: 11 , y: 22
x: 12 , y: 23
x: 13 , y: 24

```

在本例中使用友元函数重载单目运算符“++”时,形参是对象的引用,是通过传址的方法传递参数的,函数形参 op.x 和 op.y 的改变将引起实参 ob.x 和 ob.y 的变化,因而这个程序的运行结果是正确的。

一般而言,如果在类 X 中采用友元函数重载单目运算符@,而 aa 是类 X 的对象,则以下两种函数调用方法是等价的:

```

@aa;           // 隐式调用
operator@(aa); // 显式调用

```

注意:本例在 VC++ 6.0 环境下运行时,第 1 行应改为“#include<iostream.h>”,并将第 2 行删去。

说明:

(1) 运算符重载函数 operator @() 可以返回任何类型,甚至可以是 void 类型,但通常返回类型与它所操作的类类型相同,这样可使重载运算符用在复杂的表达式中。例如,在例 7.2 中,可以将几个复数连续进行加运算。

(2) 有的运算符不能定义为友元运算符重载函数,如赋值运算符=、下标运算符[]、函数调用运算符()等。

(3) C++ 编译器根据参数的个数和类型来决定调用哪个重载函数。因此,可以为同一个运算符定义几个运算符重载函数来进行不同的操作。

7.2.2 运算符重载函数作为类的成员函数

在 C++ 中, 可以把运算符函数重载为类的成员函数。

1. 运算符重载函数作为类的成员函数的语法形式

(1) 在类的内部, 将运算符函数重载为类的成员函数的格式如下:

返回类型 operator 运算符(形参表)

```
{
```

函数体

```
}
```

(2) 将运算符函数重载为类的成员函数, 也可以在类中声明成员函数的原型, 在类外定义。

在类的内部, 声明运算符函数原型的格式如下:

```
class X {
```

```
...
```

返回类型 operator 运算符(形参表);

```
...
```

```
};
```

在类外, 定义运算符函数的格式如下:

返回类型 X::operator 运算符(形参表)

```
{
```

函数体

```
}
```

其中, X 是运算符函数所在类的类名; 返回类型指定了运算符函数的返回值类型; operator 是定义运算符函数的关键字; 运算符即是要重载的运算符名称 (必须是 C++ 中可重载的运算符); 形参表中给出重载运算符所需要的参数和类型。由于运算符函数是该类的成员函数, 所以在类外定义时需要缀上 “类名::”。

在运算符函数的形参表中, 若运算符是单目的, 则参数表为空; 若运算符是双目的, 则参数表中有一个操作数。下面分别予以介绍。

2. 用成员函数重载双目运算符

对双目运算符而言, 运算符重载函数的形参表中仅有一个参数, 它作为运算符的右操作数, 此时当前对象作为运算符的左操作数, 它是通过 this 指针隐含地传递给函数的。例如:

```
class X{
...
    int operator+(X a);
...
};
```

在类 X 中声明了重载 “+” 的成员函数, 返回类型为 int, 它具有两个操作数, 一个是当前对象, 另一个是对象 a。

下面看一个采用双目成员运算符重载函数来完成例 7.2 中同样的工作的例子。

例 7.4 将运算符函数重载为成员函数, 用于进行复数的加法运算。

```
#include<iostream>
```

```

using namespace std;
class Complex{                               //声明复数类 Complex
public:
    Complex(double r=0.0, double i=0.0);    //声明构造函数
    void display();                          //显示输出复数
    Complex operator +(Complex& c);          //声明用成员函数重载运算符 "+"
private:
    double real;                             //复数的实数部分
    double imag;                             //复数的虚数部分
};
Complex::Complex(double r, double i)        //定义构造函数
{ real=r; imag=i; }
Complex Complex::operator+(Complex& c)      //重载运算符 "+" 的实现
{ Complex temp;
  temp.real=real+c.real;
  temp.imag=imag+c.imag;
  return temp;
}
void Complex::display()                     //显示复数的实数部分和虚数部分
{ cout<<real;
  if (imag>0) cout<<"+";
  if (imag!=0) cout<<imag<<"i\n";
}
int main()
{ Complex com1(2.3, 4.6), com2(3.6, 2.8), com3; //定义3个复数类对象
  com3=com1+ com2;                           //复数相加
  com1.display();                            //输出复数 com1
  com2.display();                            //输出复数 com2
  com3.display();                            //输出复数相加的结果 com3
  return 0;
}

```

程序运行结果如下:

```

2.3+4.6i
3.6+2.8i
5.9+7.4i

```

从本例可以看出,对复数重载了“+”运算符后,再进行复数运算时,只需像基本数据类型运算一样书写即可,这样给用户带来了很大的方便,并且很直观。

在主函数 main 中的语句:

```
com3= com1+ com2;
```

C++将其解释为:

```
com3= com1.operator +(com2);
```

由此我们可以看出,在进行复数的加法运算时,成员运算符重载函数 operator+ 实际上是由双目运算符左边的对象 com1 调用的,尽管双目运算符重载函数的参数表只有一个操作数 com2,但另一个操作数是由对象 com1 通过 this 指针隐含地传递的。

一般而言,如果在类 X 中采用成员函数重载双目运算符@,成员运算符重载函数

operator@所需的一个操作数由对象 aa 通过 this 指针隐含地传递, 它的另一个操作数 bb 在参数表中显示, aa 和 bb 是类 X 的两个对象, 则以下两种函数调用方法是等价的:

```
aa @ bb;                //隐式调用
aa.operator @ (bb);      //显式调用
```

3. 用成员函数重载单目运算符

对单目运算符而言, 成员运算符重载函数的参数表中没有参数, 此时当前对象作为运算符的一个操作数。

下面是一个重载单目运算符 “++” 的例子。

例 7.5 用成员函数重载单目运算符 “++”。

```
#include<iostream>
using namespace std;
class Coord{                //声明类 Coord
public:
    Coord(int i=0, int j=0);
    void display();
    Coord operator ++();      //声明用成员函数重载运算符 “++”
private:
    int x, y;
};
Coord::Coord(int i, int j)
{ x=i; y=j; }
void Coord::display()
{ cout<<"  x: "<<x<<" , y: "<<y<<endl; }
Coord Coord::operator ++()    //重载运算符 “++” 的实现
{ ++x; ++y;
  return *this;
}
int main()
{ Coord ob(11, 22);
  ob.display();
  ++ob;                //隐式调用运算符函数 operator++()
  ob.display();
  ob.operator ++();     //显式调用运算符函数 operator++()
  ob.display();
  return 0;
}
```

程序运行结果如下:

```
x: 11, y: 22
x: 12, y: 23
x: 13, y: 24
```

由于 this 指针是指向当前对象的指针, 因此语句 “return *this;” 返回的是当前对象的值, 即调用运算符重载函数 operator++ 的对象 ob 的值。

不难看出, 对类 Coord 重载了运算符++后, 对 Coord 类对象的加 1 操作变得非常方便, 就像对整型数进行加 1 操作一样。

本例主函数 main 中调用成员运算符函数 operator++ 的两种方式：

```
++ob;
```

与

```
ob.operator++();
```

是等价的，其执行效果是完全相同的。

从本例还可以看出，当用成员函数重载单目运算符时，没有参数被显式地传递给成员运算符重载函数。参数是通过 this 指针隐含地传递给函数。

一般而言，采用成员函数重载单目运算符时，以下两种方法是等价的：

```
@aa;           //隐式调用
aa.operator@(); //显式调用
```

成员运算符函数 operator @ 所需的一个操作数由对象 aa 通过 this 指针隐含地传递。因此，在它的参数表中没有参数。

7.2.3 运算符重载应该注意的几个问题

运算符重载应该注意的几个问题如下。

(1) C++ 中只能对已有的 C++ 运算符进行重载，不允许用户自己定义新的运算符。例如，虽然某些程序语言将 “**” 作为指数运算符，但是 C++ 语言编程时不能重载 “**”，因为 “**” 不是 C++ 运算符。

(2) C++ 中绝大部分的运算符允许重载，不能重载的运算符只有以下几个：

.	成员访问运算符
.*	成员指针访问运算符
::	作用域运算符
sizeof	长度运算符
?:	条件运算符

(3) 运算符重载是针对新类型数据的实际需要，对原有运算符进行适当的改造完成的。一般来讲，重载的功能应当与原有的功能相类似（如用 “+” 实现加法，用 “-” 实现减法）。从理论上说，我们可以将 “+” 运算符重载为执行减法操作，但是这样的做法违背了运算符重载的初衷，非但没有提高可读性，反而容易造成混乱。所以保持原含义，容易被接受，也符合人们的习惯。

(4) 重载不能改变运算符的操作对象（即操作数）的个数。例如，在 C++ 中，运算符 “+” 是一个双目运算符（即只能带两个操作数），重载后仍为双目运算符，需要两个参数。

(5) 重载不能改变运算符原有的优先级。C++ 已经预先规定了每个运算符的优先级，以决定运算次序。例如，C++ 规定，乘法运算符 “*” 的优先级高于减法运算符 “-” 的优先级，因此在下面表达式中，乘法运算在减法运算之前进行：

```
x=y-a*b;
```

也就是说，上列表达式等价于

```
x=y-(a*b);
```

即使我们针对某个自定义类型重载了乘法运算符 “*” 和减法运算符 “-”，我们也不能改

变这两个运算符的优先级关系，使它们按先做减法后做乘法的次序执行。如果确实需要改变某运算符的运算顺序，只能采用加括号“()”的办法进行强制改变。

(6)重载不能改变运算符原有的结合特性。例如，在C++语言中乘、除法运算符“*”和“/”都是左结合的，因此下列表达式：

```
x=a/b*c;
```

等价于 $x=(a/b)*c$;

而不等价于

```
x=a/(b*c);
```

我们无法重载运算符“*”和“/”，使它们变成右结合的。因此，必要时只能使用括号来改变它们的运算顺序。

(7)运算符重载函数的参数不能全部是C++预定义的基本数据类型，例如以下定义运算符重载函数的方法是错误的：

```
int operate+(int x, int y)
{ return x+y;}
```

这项规定的目的是，防止用户修改用于基本类型数据的运算符性质。因为，假如允许运算符重载函数的参数全部是C++基本数据类型的话，可以定义以下运算符重载函数：

```
int operate+(int x, int y)
{ return x-y;}
```

如果有表达式 $5+3$ ，它的结果是8呢？还是2呢？显然，这是绝对不允许的。

(8)双目运算符一般可以被重载为友元函数或成员函数，但有一种情况，必须使用友元函数。

例如，如果将一个类AB的对象与一个整数相加，可用成员函数重载运算符“+”：

```
AB::operator +(int x)
{ AB temp;
  temp.a=a+x;
  temp.b=b+x;
  return temp;
}
```

若ob1和ob2是类AB的对象，则以下语句是正确的：

```
ob2=ob1+200;
```

这条语句被C++编译系统解释为

```
ob2=ob1.operator(200);
```

由于对象ob是运算符“+”的左操作数，所以它可以调用“+”运算符重载函数operator+()，执行结果是对象ob1数据成员a和b都被加上一个整数200。然而，以下语句就不能工作了：

```
ob2=200+ob1; //编译错误，运算符“+”的左侧是整数
```

这条语句被C++编译系统解释为

```
ob2=200.operator(ob1);
```

由于运算符“+”的左操作数是一个整数200，而不是该类的对象，编译时将出现错误，

因为整数 200 不能调用成员函数。

如果定义以下的两个友元函数：

```
friend AB operator +(AB ob, int x);    //声明用友元函数重载运算符 "+"
//运算符 "+" 的左侧是类对象, 右侧是整数
friend AB operator +(int x, AB ob);    //声明用友元函数重载运算符 "+"
//运算符 "+" 的左侧是整数, 右侧是类对象
```

当类 AB 的一个对象与一个整数相加时, 无论整数出现在左侧还是右侧, 使用友元运算符重载函数都能得到很好的解决。这就解决了使用成员函数时, 由于整数出现在运算符 "+" 的左侧而出现的错误。下述例子就说明了实现的具体方法。

例 7.6 使用友元运算符重载函数解决对象与整数相加的问题。

```
#include<iostream>
using namespace std;
class AB{                                //声明类 AB
public:
    AB(int x=0, int y=0);
    friend AB operator +(AB ob, int x);
    // 声明用友元函数重载运算符 "+", 运算符的左侧是类对象, 右侧是整数
    friend AB operator +(int x, AB ob);    //声明友元运算符函数
    // 声明用友元函数重载运算符 "+", 运算符的左侧是整数, 右侧是类对象
    void show();
private:
    int a, b;
};
AB::AB(int x, int y)
{ a=x; b=y; }
AB operator +(AB ob, int x)              //重载运算符 "+" 的实现,
//运算符的左侧是类对象, 右侧是整数

{ AB temp;
  temp.a=ob.a+x;
  temp.b=ob.b+x;
  return temp;
}
AB operator +(int x, AB ob)              // 重载运算符 "+" 的实现
// 运算符的左侧是整数, 右侧是类对象

{ AB temp;
  temp.a=x+ob.a;
  temp.b=x+ob.b;
  return temp;
}
void AB::show()
{ cout<<"a="<<a<<" "<<"b="<<b<<"\n"; }
int main()
{ AB ob1(50, 60), ob2;
  ob2=ob1+20;
  ob2.show();
  ob2=40+ob1;
  ob2.show();
  return 0;
}
```

程序运行结果如下：

```
a=70 b=80
a=90 b=100
```

7.3 前置运算符和后置运算符的重载

我们知道，自增运算符“++”和自减运算符“--”放置在变量的前面与后面，其作用是有区别的，例如：

```
int i=10;
int x;
x=i++;           //后置方式，把 i 原值赋给 x，然后 i 加 1
x=++i;           //前置方式，先把 i 加 1，再把 i 值赋给 x
```

但是，早期版本的 C++在重载“++”或“--”时，不能显式地区分是前置方式还是后置方式。也就是说，在例 7.5 的 main 函数中，以下两条语句是完全相同的：

```
ob++;
```

与 ++ob;

在 C++ 2.1 及以后的版本中，编辑器可以通过在运算符函数参数表中是否插入关键字 int 来区分这两种方式。

对于前置方式++ob，可以用运算符函数重载为

```
ob.operator ++();           //用成员函数重载
```

或

```
operator ++(X& ob);         //用友元函数重载，其中 ob 为类 X 对象的引用
```

对于后置方式 ob++，可以用运算符函数重载为

```
ob.operator ++(int);        //用成员函数重载
```

或

```
operator++(X& ob, int);     //用友元函数重载
```

这里的 int 类型参数只是用来区别后置++与前置++，此外没有任何其他的作用。

例 7.7 使用成员函数分别以前置方式和后置方式重载运算符“++”。

```
#include<iostream>
using namespace std;
class Three_d{
public:
    Three_d(int T1=0, int T2=0, int T3=0);    //声明构造函数
    void disp();
    Three_d operator++();                     //声明自增运算符“++”重载成员函数(前置方式)
    Three_d operator++(int);                 //声明自增运算符“++”重载成员函数(后置方式)
private:
    int t1, t2, t3;
```

```

};
Three_d::Three_d(int T1, int T2, int T3)    //定义构造函数
{ t1=T1; t2=T2; t3=T3;}
void Three_d::disp()                        //输出数据成员的值
{ cout<<"t1: "<<t1<<" t2: "<<t2<<" t3: "<<t3<<endl;}
Three_d Three_d::operator++()              //定义自增运算符“++”重载成员函数(前置方式)
{ ++t1; ++t2; ++t3;
  return *this;                            //返回自增后的当前对象
}
Three_d Three_d::operator++(int)            //定义自增运算符“++”重载成员函数(后置方式)
{ Three_d temp(*this);
  t1++; t2++; t3++;
  return temp;                             //返回自增前的当前对象
}
int main()
{ Three_d obj1(4, 5, 6), obj2;
  cout<<"obj1的原值      : ";
  obj1.disp();                             //显示 obj1 的原值
  ++obj1;                                  //调用 operator++() (前置方式)
  cout<<"执行++obj1后的 obj1 的值: ";
  obj1.disp();                             //显示执行++obj1后的 obj1 的值
  obj2=obj1++;                             //将自增前的对象 obj1 的值赋给 obj2
  cout<<"执行 obj1++前的 obj1 的值: ";
  obj2.disp();                             //显示 obj2 保存的是执行 obj1++前的 obj1 的值
  cout<<"执行 obj1++后的 obj1 的值: ";
  obj1.disp();                             //显示执行 obj1++后的 obj1 的值
  return 0;
}

```

请注意前置自增运算符“++”和后置自增运算符“++”之间的区别。前者是先自增，返回的是修改后的对象本身。后者返回的是自增前的对象，然后对象自增。请仔细分析以下程序运行结果。

程序运行结果如下：

```

obj1 的原值      : t1: 4 t2: 5 t3: 6
执行++obj1 后的 obj1 的值: t1: 5 t2: 6 t3: 7
执行 obj1++前的 obj1 的值: t1: 5 t2: 6 t3: 7
执行 obj1++后的 obj1 的值: t1: 6 t2: 7 t3: 8

```

可以看出，重载后置自增运算符时，多了一个 `int` 型的参数，这个参数只是为了与前置自增运算符重载函数有所区别，此外没有任何作用，在定义函数时也不必使用此参数，因此可不写参数名，只需在括号中写 `int` 即可。C++编译系统在遇到后置自增运算符时，会自动调用此函数，参数 `int` 一般被传递值 0。例如表达式 `obj++` 就相当于函数调用 `obj.operator++(0)`。

例 7.8 使用友元函数分别以前置方式和后置方式重载了运算符“—”。

```

#include<iostream>
using namespace std;
class Three_d{

```

```

public:
    Three_d(int T1=0, int T2=0, int T3=0);    //声明构造函数
    void disp();
    friend Three_d operator--(Three_d &);    //声明自减运算符 "--" 重载友元函数 (前置方式)

    friend Three_d operator--(Three_d &, int);    //声明自减运算符 "--" 重载友元函数 (后置方式)

private:
    int t1, t2, t3;
};

Three_d::Three_d(int T1, int T2, int T3)    //定义构造函数
{ t1=T1; t2=T2; t3=T3; }
void Three_d::disp()    //输出数据成员的值
{ cout<<"t1: "<<t1<<" t2: "<<t2<<" t3: "<<t3<<endl; }
Three_d operator--(Three_d &op)    //定义自减运算符 "--" 重载友元函数 (前置方式)
{ --op.t1; --op.t2; --op.t3;
  return op;
}
Three_d operator--(Three_d &op, int)    //定义自减运算符 "--" 重载友元函数 (后置方式)
{ op.t1--; op.t2--; op.t3--;
  return op;
}
int main()
{ Three_d obj1(4, 5, 6), obj2;
  cout<<"obj1的原值      : ";
  obj1.disp();    //显示 obj1 的原值
  --obj1;    //调用 operator-- (Three_d&) (前置方式)
  cout<<"执行--obj1后的 obj1 的值: ";
  obj1.disp();    //显示执行--obj1后的 obj1 的值
  obj1--;    //调用 operator-- (Three_d&, int) (后置方式)
  cout<<"执行obj1--后的 obj1 的值: ";
  obj1.disp();    //显示执行obj1--后的 obj1 的值
  return 0;
}

```

程序运行结果如下:

```

obj1 的原值      : t1:4 t2:5 t3:6
执行--obj1 后的 obj1 的值: t1:3 t2:4 t3:5
执行obj1--后的 obj1 的值: t1:2 t2:3 t3:4

```

说明:

由于友元运算符重载函数没有 this 指针, 所以不能引用 this 指针所指的对象。使用友元函数重载自增运算符“++”或自减运算符“--”时, 应采用对象引用参数传递数据。例如:

```

friend Three_d operator ++(Three_d&);    //前置方式
friend Three_d operator ++(Three_d&, int);    //后置方式

```

可以看出, 重载后置自减运算符时, 多了一个 int 型的参数, 这个参数只是为了与前置自增

运算符重载函数有所区别, 此外没有任何作用, 在定义函数时也不必使用此参数, 因此可不必写参数名, 只需在括号中写 `int` 即可。C++编译系统在遇到后置自减运算符时, 会自动调用此函数, 参数 `int` 一般被传递给值 0。例如, 表达式 `obj1--` 就相当于函数调用 `oprator--(obj1, 0)`。

7.4 重载插入运算符和提取运算符

前面我们介绍了系统预定义的基本数据类型的输入或输出。对于用户自定义类型(类类型、结构体类型等)的输入或输出, 在 C++中可以通过重载运算符 “>>” 和 “<<” 来实现。

7.4.1 重载插入运算符 “<<”

在 C++中, “<<” 本来是被定义为左移位运算符, 由于在 `iostream` 头文件中对它进行了重载, 使它能用作基本类型数据的输出运算符。插入运算符(也称输出运算符)“<<”是一个双目运算符, 有两个操作数, 左操作数为输出流类 `ostream` 的一个流对象(如 `cout`, 将在第 9 章详细介绍), 右操作数为一个系统预定义的基本类型的常量或变量。在头文件 `iostream` 中有一组运算符函数对运算符 “<<” 进行重载, 以便能用它输出各种标准类型的数据, 其原型具有

```
ostream& operator<<( ostream& 类型名 );
```

的形式。

其中, 类型名是指 `int`、`float`、`double`、`char*`、`char` 等 C++基本类型。这表明, 只要输出的数据属于其中的一种, 就可以直接使用插入运算符 “<<” 完成基本类型数据的输出任务。

例如当系统执行

```
cout<<"This is a string.\n";
```

操作时, 就调用了插入运算符重载函数

```
ostream& operator<<( ostream& char* );
```

以上语句相当于

```
cout.operator<<("This is a string.\n");
```

它的功能是将字符串 “This is a string.” 插入到流对象 `cout` 中, `cout` 为标准输出流对象, 它与标准输出设备(通常为显示器)连在一起。于是在显示器屏幕上显示出字符串 “This is a string.”。

C++对插入运算符 “<<” 的功能进行了扩充, 可以通过重载运算符 “<<” 实现用户自定义类型的输出。定义插入运算符 “<<” 重载函数的一般格式如下:

```
ostream &operator<<(ostream &out, 自定义类名& obj)
```

```
{
    out<<obj.item1;
    out<<obj.item2;
    ...
    out<<obj.itemn;
    return out;
}
```

函数中第 1 个参数 out 是 ostream 类对象的引用。这意味着 out 必须是输出流对象，它可以是其他任何正确的标识符，但必须与 return 后面的标识符相同。第 2 个参数 obj 为用户自定义类型的对象的引用。item1, ..., itemn 为用户自定义类型中的数据成员。插入运算符重载函数不能是所操作的类的成员函数，但可以是该类的友元函数或普通函数。

下面看一个插入运算符“<<”重载的例子。

例 7.9 将运算符“+”重载为友元函数，用于进行复数的加法运算，并用重载的运算符“<<”输出复数。

```
#include<iostream>
using namespace std;
class Complex{                               //声明复数类 Complex
public:
    Complex(double r=0.0, double i=0.0);    //声明构造函数
    friend Complex operator +(Complex& a, Complex& b); //声明运算符“+”重载为友元函数
    friend ostream& operator <<(ostream& , Complex& ); //声明运算符“<<”重载为友元函数
private:
    double real;                             //复数实部
    double imag;                             //复数虚部
};
Complex::Complex(double r, double i)        //定义构造函数
{ real=r; imag=i; }
Complex operator+(Complex& a, Complex& b)    //定义运算符“+”重载函数
{ Complex temp;
  temp.real=a.real+b.real;
  temp.imag=a.imag+b.imag;
  return temp;
}
ostream& operator <<(ostream& out , Complex& com) //定义运算符“<<”重载函数
{ out<<com.real;
  if (com.imag>0) out<<"+";
  if (com.imag!=0) out<<com.imag<<"i\n";
  return out;
}
int main()
{ Complex com1(2.3, 4.6), com2(3.6, 2.8), com3; //定义 3 个复数类对象
  cout<<com1;                                //输出复数 com1
  cout<<com2;                                //输出复数 com2
  com3=com1+com2;                            //复数相加
  cout<<com3;                                //输出复数相加结果 com3
  return 0;
}
```

程序运行结果如下：

```
2.3+4.6i
3.6+2.8i
5.9+7.4i
```

可以看到在对运算符“<<”重载后，在程序中用“<<”不仅能输出基本类型数据，而且可以输出用户自己定义的类对象。用“cout<<com1;”就能以复数的形输出复数对象 com1 的值。下面对插入运算符“<<”重载的实现作一些说明。

程序中运算符“<<”重载函数中的形参 out 是 ostream 类对象的引用，形参名 out 是用户任意起的。主函数中语句

```
cout<<com1;
```

运算符“<<”的左边是 ostream 类对象 cout，右边的是 Complex 类对象 com1。由于已将运算符“<<”的重载函数声明为类 Complex 的友元函数，C++编译系统把“cout<<com1;”解释为

```
operator<<(cout, com1);
```

即以 cout 和 com1 作为实参，调用下面的运算符“<<”重载函数：

```
ostream& operator <<(ostream& out, Complex& com)
{ out<<com.real;
  if (com.imag>0) out<<"+";
  if (com.imag!=0) out<<com.imag<<"i\n";
  return out;
}
```

调用函数时，形参 out 成为 cout 的引用，形参 com 成为 com1 的引用，因此调用的过程相当于执行：

```
cout<<com.real;
if (com.imag>0) cout<<"+";
if (com.imag!=0) cout<<com.imag<<"i\n";
return cout;
```

于是执行“cout<<com1;”，输出：

```
2.3+4.6i
```

说明：

在 VC++ 6.0 环境下运行时，第 1 行应改为“#include<iostream.h>”，并将第 2 行删去。

7.4.2 重载提取运算符“>>”

在 C++中，“>>”本来是被定义为右位移运算符，由于在 iostream 头文件对它进行了重载，使它能用作基本类型数据的输入运算符。插入运算符（也称输入运算符）“>>”也是一个双目运算符，有两个操作数，左面的操作数是输入流类 istream 的一个对象（如 cin），右面的操作数是系统预定义的任何基本数据类型的变量。在头文件 iostream 中也有一组提取运算符函数对运算符“>>”进行重载，以便能用它输入各种标准类型的数据，其原型具有

```
istream& operator>>(istream& 类型名 &);
```

的形式。

其中，类型名也是指 int、float、double、char*、char 等 C++标准类型。这表明，只要输入的数据属于其中的一种，就可以直接使用提取运算符“>>”完成标准类型数据的输入任务。例如当系统执行

```
int x;
cin>>x;
```

操作时, 将根据实参 x 的类型调用相应的提取运算符重载函数, 并把 x 传送给对应的形参, 接着从标准输入流对象 cin (它与标准输入设备连在一起, 通常为键盘) 读入一个值并赋给 x (因为形参是 x 的引用)。

C++对提取运算符 “>>” 的功能进行了扩充, 可以通过重载运算符 “>>” 实现用户自定义类型的输入。定义提取运算符函数与插入运算符函数的格式基本相同, 只是要把 ostream 换成 istream , 把 “<<” 用 “>>” 代替。完整的格式如下:

```
istream& operator>>(istream& in, 自定义类名& obj)
{ in>>obj.item1;
  in>>obj.item2;
  ...
  in>>obj.itemn;
  return in;
}
```

函数中第 1 个参数 in 是 istream 类对象的引用。这意味着 in 必须是输入流对象, 它可以是其他任何正确的标识符, 但必须与 return 后面的标识符相同。第 2 个参数 obj 为用户自定义类型的对象的引用。item1, ..., itemn 为用户自定义类型中的数据成员。

与插入运算符重载函数一样, 提取运算符重载函数也不能是所操作的类的成员函数, 但可以是该类的友元函数或普通函数。下面举例说明。

例 7.10 重载运算符 “<<” 和 “>>”, 使用户能直接输入和输出复数。

```
#include<iostream>
using namespace std;
class Complex{ //声明复数类 Complex
public:
    Complex(double r=0.0, double i=0.0); //声明构造函数
    friend ostream& operator<<(ostream& , Complex& ); // 声明运算符 “<<” 重载为友元函数
    friend istream& operator>>(istream& , Complex&); //声明运算符 “>>” 重载为友元函数
private:
    double real; //复数实部
    double imag; //复数虚部
};
Complex::Complex(double r, double i) //定义构造函数
{ real=r; imag=i;}
ostream& operator <<(ostream& out , Complex& com)
{ out<<com.real; //定义运算符 “<<” 重载函数
  if (com.imag>0) out<<"+";
  if (com.imag!=0) out<<com.imag<<"i\n";
  return out;
}
istream& operator>>(istream& in, Complex& com) //定义运算符 “>>” 重载函数
{ cout<<" Enter the real part and imaginary part of complex number:\n";
  cout<<"real:";
  in>>com.real;
  cout<<"imag:";
```



```

        in>>com.imag;
        return in;
    }

    int main()
    { Complex com1;                                // 定义复数类对象 com1
      cin>>com1;                                    // 输入复数 com1 的值
      cout<<com1;                                  // 输出复数 com1 的值
      return 0;
    }

```

程序运行结果如下:

Enter the real part and imaginary part of complex number:

real:1.1✓

imag:2.2✓

1.1+2.2i

在这个程序中,定义了插入运算符函数和提取运算符函数,它们都是类 Complex 的友元函数,分别完成对该类对象的输出和输入操作。读者可以参照例 7.9 题自行分析。

说明:

在 VC++ 6.0 环境下运行时,第 1 行应改为“#include<iostream.h>”,并将第 2 行删去。

7.5 不同类型数据间的转换

7.5.1 系统预定义类型间的转换

类型转换是将一种类型的值转换为另一种类型值。对于系统预定义的基本类型(如 int、float、double、char 等),C++ 提供两种类型转换方式,一种是隐式类型转换,另一种是显式类型转换。

1. 隐式类型转换

在 C++ 中,某些不同类型的数据之间可以自动转换,例如:

```

int x=5, y;
y=3.5+x;

```

C++ 编译系统对 3.5 是作为 double 型数据处理的,在进行“3.5+x”时,先将 x 的值 5 转换成 double 型,然后与 3.5 相加,得到的和为 8.5,在向整型变量 y 赋值时,将 8.5 转换成整数 8,然后赋给 y。这种转换是由 C++ 编译系统自动完成的,用户不需干预,称为隐式类型转换。

2. 显式类型转换

编程人员在程序中可以明确地指出将一种数据类型转换成另一种指定的类型,这种转换称为显式类型转换。显式类型转换常采用下述方法表示:

(1) C 语言中采用的形式为:

(类型名)表达式

例如:

```
double i=2.2, j=3.2
```

```
cout<<(int)(i+j);
```

将表达式 $i+j$ 的值 5.4 强制转换成整数 5 后输出。

(2) C++语言中采用的形式为:

类型名(表达式)

例如:

```
double i=2.2, j=3.2;
cout<<(int)(i+j);
```

此时,也将表达式 $i+j$ 的值 5.4 强制转换成整数 5 后输出。

C++保留了 C 语言的用法,但提倡采用 C++提供的方法。

7.5.2 类类型与系统预定义类型间的转换

前面介绍的是系统预定义的标准数据类型之间的转换。那么,对于用户自己定义的类型而言,如何实现它们与其他数据类型之间的转换呢?通常,可归纳为以下二种方法:

(1) 通过转换构造函数进行类型转换。

(2) 通过类型转换函数进行类型转换。

下面分别予以介绍。

1. 转换构造函数

转换构造函数也是构造函数的一种,它具有类型转换的作用,它的作用是将一个其他类型的数据转换成它所在类的对象。我们先回顾一下例 7.2 将运算符函数重载为友元函数,用于进行复数的加法运算的例子。在这个例子中,定义了一个将两个复数相加的友元运算符重载函数,在主函数

```
int main( )
{ Complex com1(1.1, 2.2), com2(3.3, 4.4), com3;
  com3=com1+com2;           //复数类 Complex 对象相加
  com3.print();
  return 0;
}
```

中执行语句“ $com3=com1+com2;$ ”时,调用这个运算符重载函数,将 Complex 类的对象 $com1$ 和 $com2$ 相加,程序运行结果如下:

```
4.4+6.6i
```

如果我们主函数 main 中的语句“ $com3=com1+com2;$ ”改成

```
com3= com1 + 7.7;
```

编译时将出现错误,因为在这个例子中不能将一个 Complex 类的对象 $com1$ 和一个 double 类型的数据 7.7 相加。

为了实现将一个 Complex 类的对象和一个 double 类型的数据相加,可以定义以下转换构造函数:

```
Complex(double r)           //转换构造函数
{ real=r; imag=0;
}
```

其作用是将 double 类型的参数 r 转换成 Complex 类的对象,将 r 的值赋给复数对象的实

部，虚部为 0。

转换构造函数只有一个形参，用户可以根据需要定义转换构造函数，在函数体中告诉编译系统怎样去进行类型转换。

假如在 `Complex` 类中定义了上面的转换构造函数，将主函数定义为：

```
int main()
{ Complex com1(1.1, 2.2), com3;
  Complex com2(7.7);           //调用转换构造函数，将 7.7 转换成对象 com2
  com3=com1+com2;              //复数对象相加
  com3.print();
  return 0;
}
```

执行语句“`Complex com2(7.7);`”时，将会调用所定义的转换构造函数，将 `double` 类型的数据 7.7 转换成 `Complex` 类的对象 `com2`（其 `real` 的值为 7.7，`imag` 的值为 0），然后执行语句“`com3=com1+com2;`”，调用友元运算符重载函数将 `Complex` 类的对象 `com1` 和 `com2` 相加，程序运行结果如下：

8.8+2.2i

下面是这个例子的完整程序。

例 7.11 转换构造函数的应用。

```
#include<iostream>
using namespace std;
class Complex{
public:
  Complex(){} //不带参数的构造函数
  Complex(double r, double i) //带两个参数的构造函数
  { real=r; imag=i; }
  Complex(double r) //转换构造函数
  { real=r; imag=0; }
  friend Complex operator+(Complex &col, Complex &co2); //声明友元运算符重载函数

  void print();
private:
  double real; //复数的实部
  double imag; //复数的虚部
};

Complex operator+(Complex &col, Complex &co2) //定义友元运算符重载函数
{ Complex temp;
  temp.real=col.real+co2.real;
  temp.imag=col.imag+co2.imag;
  return temp;
}

void Complex::print() //输出复数对象的值
{ cout<<real;
  if (imag>0) cout<<"+";
  if (imag!=0) cout<<imag<<"i\n";
}
```

```
int main( )
{ Complex com1(1.1, 2.2), com3;
  Complex com2(7.7);           //调用转换构造函数, 将 7.7 转换成对象 com2
  com3=com1+com2;               //两个 Complex 类对象相加
  com3.print();                 //输出复数对象的值
  return 0;
}
```

程序运行结果如下:

8.8+2.2i

在这个例子中, 通过调用转换构造函数将 `double` 类型的数据 7.7 转换成名为 `com2` 的 `Complex` 类对象, 然后执行语句 “`com3=com1+com2;`”, 将两个 `Complex` 类对象相加。我们也可以将上例主函数中的两条语句

```
Complex com2(7.7);           //调用转换构造函数, 将 7.7 转换成对象 com2
com3=com1+com2;               //两个 Complex 类对象相加
```

修改成

```
com3=com1+Complex(7.7);
```

其中执行 “`Complex(7.7)`” 时, 调用了转换构造函数, 将 `double` 类型的 7.7 转换成一个无名的 `Complex` 类临时对象(其值为 $7.7+0i$), 然后将此无名的临时对象与 `Complex` 类对象 `com1` 相加, 相加的结果赋给了 `Complex` 类对象 `com3`。其运行结果仍然是:

8.8+2.2i

通常, 使用转换构造函数将一个指定的数据转换为类对象的方法如下。

(1) 先声明一个类 (例如 `Complex`)。

(2) 在这个类中定义一个只有一个参数的构造函数, 参数是待转换类型的数据, 在函数体中指定转换的方法。

例如:

```
Complex(double r)
{ real=r; imag=0;}           //函数体中指定转换的方法
```

(3) 可以用以下形式进行类型转换:

类名(待转换类型的数据)

例如:

```
Complex(7.7)
```

这时, C++系统就自动调用转换构造函数, 将 `double` 类型的 7.7 转换成 `Complex` 类型的无名临时对象, 其值为:

```
临时对象.real = 7.7
临时对象.imag = 0
```

说明:

(1) 转换构造函数也是一种构造函数, 它遵循构造函数的一般规则。转换构造函数只有一个参数, 作用是将一个其他类型的数据转换成它所在类的对象。但是, 有一个参数的构造函数不一定是转换构造函数, 它可以是普通的构造函数, 仅仅起对象初始化的作用。

(2) 转换构造函数不仅可以将一个系统预定义的基本类型数据转换成类的对象，也可以将另一个类的对象转换成转换构造函数所在的类对象。需要深入了解的读者可以参阅有关书籍，在此不作详细介绍。

2. 类型转换函数

通过转换构造函数可以将一个指定类型的数据转换为类的对象。但是不能反过来将一个类的对象转换成其他类型的数据，例如不能将一个 `Complex` 类的对象转换成 `double` 类型的数据。为此，C++ 提供了一个称为类型转换函数的函数来解决这个转换问题。类型转换函数的作用是将一个类的对象转换成另一类型的数据。在类中，定义类型转换函数的一般格式为：

```
operator 目标类型()
{
    函数体
}
```

其中，目标类型为希望转换成的类型名，它既可以是预定义的基本数据类型也可以是其他类的类型。类型转换函数的函数名为“`operator 目标类型`”，在函数名前面不能指定函数返回类型，也不能有参数。通常，其函数体的最后一条语句是 `return` 语句，返回值的类型是该函数的目标类型。例如，已经声明了一个 `Complex` 类，可以在 `Complex` 类中定义一个类型转换函数：

```
operator double()
{ return real;
}
```

这个类型转换函数的函数名是“`operator double`”，希望转换成的目标类型为 `double`，函数体为“`return real;`”。这个类型转换函数的作用是将 `Complex` 类对象转换为一个 `double` 类型的数据，其值是 `Complex` 类中的数据成员 `real` 的值。

下面看一个利用这个类型转换函数进行类型转换的例子。

例 7.12 类型转换函数的应用。

```
#include<iostream>
using namespace std;
class Complex{
public:
    Complex(double r=0, double i=0) //构造函数
    { real=r;
      imag=i;
    }
    operator double()                //类型转换函数
    { return real;                    //将 Complex 类的对象转换为一个 double 类型的数据
    }
private:
    double real, imag;
};

int main()
{ Complex com(2.2, 4.4);             //定义 Complex 类的对象 com
  cout<<"Complex 类的对象 com 转换成 double 型的数据为:";
  cout<<double(com)<<endl;          //调用类型转换函数，将转换后的 double 类型的数据显示出来

  return 0;
```

}

程序运行结果如下：

Complex 类的对象 com 转换成 double 型的数据为：2.2

在以上程序中，主函数内语句“cout<<double(com)<<endl;”中的“double(com)”调用了类型转换函数，将类 Complex 的对象 com 转换成 double 类型的数据 2.2。

关于类型转换函数，有以下几点注意事项。

- (1) 类型转换函数只能定义为一个类的成员函数而不能定义为类的友元函数。类型转换函数也可以在类体中声明函数原型，而将函数体定义在类的外部。
- (2) 类型转换函数既没有参数，也不能在函数名前面指定函数返回类型。
- (3) 类型转换函数中必须有 return 语句，即必须送回目标类型的数据作为函数的返回值。
- (4) 一个类可以定义多个类型转换函数。C++编译器将根据类型转换函数名自动地选择一个合适的类型转换函数予以调用。

7.6 应用举例

例 7.13 用友元函数和成员函数两种形式实现对复数的加、减、乘、除，以及求负运算，并且重载插入运算符和提取运算符以实现复数的输入和输出。

```
#include<iostream>
using namespace std;
class Complex_A{                                //定义复数类 Complex_A
public:
    Complex_A(double r=0, double i=0)          //定义带参数的构造函数
    { x=r; y=i; }
    Complex_A operator+( Complex_A&);          //运算符“+”重载为成员函数
    Complex_A operator-( Complex_A&);          //运算符“-”重载为成员函数
    Complex_A operator*( Complex_A&);          //运算符“*”重载为成员函数
    Complex_A operator/( Complex_A&);          //运算符“/”重载为成员函数
    Complex_A operator-();                      //单目运算符“-”重载为成员函数
    friend ostream& operator<<(ostream& output, Complex_A&);
                                                //运算符“<<”重载为友元函数
    friend istream& operator>>(istream& input, Complex_A& );
                                                //运算符“>>”重载为友元函数
private:
    double x, y;
};
class Complex_B {                                //定义复数类 Complex_B
public:
    Complex_B(double r=0, double i=0)          //定义带参数的构造函数
    { x=r; y=i; }
    friend Complex_B operator+( Complex_B&, Complex_B&);
                                                //运算符“+”重载为友元函数
    friend Complex_B operator-( Complex_B&, Complex_B&);
```

```

//运算符 "-" 重载为友元函数
friend Complex_B operator*( Complex_B&, Complex_B&);
//运算符 "+" 重载为友元函数
friend Complex_B operator/( Complex_B&, Complex_B&);
//运算符 "/" 重载为友元函数
friend Complex_B operator-( Complex_B&);
//单目运算符 "-" 重载为友元函数
friend ostream& operator<<(ostream& output, Complex_B&);
//运算符 "<<" 重载为友元函数
friend istream& operator>>(istream& input, Complex_B& );
//运算符 ">>" 重载为友元函数

private:
    double x, y;
};

Complex_A Complex_A::operator+( Complex_A& com) //定义运算符 "+" 重载为成员函数
{ return Complex_A(x+com.x, y+com.y); }
}
Complex_A Complex_A::operator-( Complex_A& com) //定义运算符 "-" 重载为成员函数
{ return Complex_A(x-com.x, y-com.y); }
}
Complex_A Complex_A::operator*( Complex_A& com) //定义运算符 "*" 重载为成员函数
{ return Complex_A(x*com.x-y*com.y, x*com.x+y*com.y); }
}
Complex_A Complex_A::operator/( Complex_A& com) //定义运算符 "/" 重载为成员函数
{ double temp=com.x*com.x+com.y*com.y;
  return Complex_A((x*com.x+y*com.y)/temp, (x*com.x-y*com.y)/temp);
}
Complex_A Complex_A::operator-() //定义单目运算符 "-" 重载为成员函数
{ return Complex_A(-x, -y); }
Complex_B operator +( Complex_B& a, Complex_B& b) //定义运算符 "+" 重载为友元函数
{ return Complex_B(a.x+b.x, a.y+b.y); }
Complex_B operator-( Complex_B& a, Complex_B& b) //定义运算符 "-" 重载为友元函数
{ return Complex_B(a.x-b.x, a.y-b.y); }
Complex_B operator*( Complex_B& a, Complex_B& b) //定义运算符 "*" 重载为友元函数
{ return Complex_B(a.x*b.x-a.y*b.y, a.x*b.x+a.y*b.y); }
Complex_B operator/( Complex_B& a, Complex_B& b) //定义运算符 "/" 重载为友元函数
{ double temp=b.x*b.x+b.y*b.y;
  return Complex_B((a.x*b.x+a.y*b.y)/temp, (a.x*b.x-a.y*b.y)/temp);
}
Complex_B operator-( Complex_B& com) //定义单目运算符 "-" 重载为友元函数
{ return Complex_B(-com.x, -com.y); }

```

```

}
ostream& operator<<(ostream& output, Complex_A& com)
//定义运算符 "<<" 重载为友元函数
{
    output<<com.x;
    if (com.y>0) output<<"+";
    if (com.y !=0) output<<com.y<<"i";
    return output;
}

istream& operator>>(istream& input, Complex_A&com)
//定义运算符 ">>" 重载为友元函数
{
    cout<<"请输入复数的实部和虚部:"<<endl;
    input>>com.x>>com.y;
    return input;
}

ostream& operator<<(ostream& output, Complex_B& com)
//定义运算符 "<<" 重载为友元函数
{
    output<<com.x;
    if (com.y>0) output<<"+";
    if (com.y !=0) output<<com.y<<"i";
    return output;
}

istream& operator>>(istream& input, Complex_B& com)
//定义运算符 ">>" 重载为友元函数
{
    cout<<"请输入复数的实部和虚部:"<<endl;
    input>>com.x>>com.y;
    return input;
}

int main()
{
    Complex_A a(21, 3), b(5, 19), e;
    Complex_B c(36, 9), d(15, 3), f;
    cin>>e;
    cin>>f;
    cout<<"a="<<a<<" "<<"b="<<b<<endl;
    cout<<"c="<<c<<" "<<"d="<<d<<endl;
    cout<<"e="<<e<<" "<<"f="<<f<<endl;
    cout<<"a+b="<<a+b<<endl;
    cout<<"c+d="<<c+d<<endl;
    cout<<"a-b="<<a-b<<endl;
    cout<<"c-d="<<c-d<<endl;
    cout<<"a*b="<<a*b<<endl;
    cout<<"c*d="<<c*d<<endl;
    cout<<"a/b="<<a/b<<endl;
    cout<<"c/d="<<c/d<<endl;
    cout<<"-a="<<-a<<endl;
    cout<<"-c="<<-c<<endl;
    return 0;
}

```

程序运行结果如下:

请输入复数的实部和虚部:

1.1 2.2

请输入复数的实部和虚部:


```

3.3 4.4
a=21+3i b=5+19i
c=36+9i d=15+3i
e=1.1+2.2i f=3.3+4.4i
a+b=26+22i
c+d=51+12i
a-b=16-16i
c-d=21+6i
a*b=48+162i
c*d=513+567i
a/b=0.419689+0.124352i
c/d=2.42308+2.19231i
-a=-21-3i
-c=-36-9i

```

说明:

在 VC++ 6.0 环境下运行时, 第 1 行应改为 “#include<iostream.h>”, 并将第 2 行删去。

实 验

实验目的和要求

学习运算符重载的定义和使用方法。

实验内容和步骤

1. 分析并调试下列程序, 写出程序的输出结果, 并分析输出结果。

```

//test7_1.cpp
#include<iostream>
using namespace std;
class A {
public:
    A(int i):x(i)
    {
    }
    A()
    { x=0; }
    friend A operator ++(A a);
    friend A operator --(A &a);
    void print();
private:
    int x;
};
A operator++(A a)
{ ++a.x;
  return a;
}
A operator --(A &a)

```

```

{ --a.x;
  return a;
}
void A::print()
{ cout<<x<<endl;
}
int main()
{ A a(7);
  ++a;
  a.print();
  --a;
  a.print();
  return 0;
}

```

2. 编写一个程序，其中设计一个时间类 `Time`，用来保存时、分、秒等私有数据成员，通过重载操作符“+”实现两个时间的相加。要求将小时范围限制在大于等于0，分钟范围限制在0~59分，秒钟范围限制在0~59秒。

【提示】

时间类 `Time` 的参考框架如下：

```

class Time{
public:
    Time(int h=0, int m=0, int s=0);           //构造函数
    Time operator+(Time&);                     //运算符重载函数，实现两个时间的相加
    void disptime(string);                     //输出时间函数
private:
    int hours;                                 //小时
    int minutes;                               //分钟
    int seconds;                               //秒钟
};

```

3. 编写一个程序，要求：

(1) 声明一个类 `Complex`，定义类 `complex` 的两个对象 `c1` 和 `c2`，对象 `c1` 通过构造函数直接指定复数的实部和虚部（类的私有数据成员为 `double` 类型的 `real` 和 `imag`）为 2.5 及 3.7，对象 `c2` 通过构造函数指定复数的实部和虚部为 4.2 及 6.5；

(2) 定义友元运算符重载函数，它以 `c1`、`c2` 对象为参数，调用该函数时能返回两个复数对象相加操作；

(3) 定义成员函数 `print`，调用该函数时，能以格式“(real, imag)”输出当前对象的实部和虚部，例如：对象的实部和虚部分别是 4.2 和 6.5，则调用 `print` 函数输出格式为：(4.2, 6.5)；

(4) 编写主程序，计算出复数对象 `c1` 和 `c2` 相加的结果，并将其结果输出。

习 题

【7.1】什么是运算符重载？

【7.2】有关运算符重载正确的描述是 ()。

- A. C++语言允许在重载运算符时改变运算符的操作个数
- B. C++语言允许在重载运算符时改变运算符的优先级
- C. C++语言允许在重载运算符时改变运算符的结合性
- D. C++语言允许在重载运算符时改变运算符的原来的功能

【7.3】以下能用友元函数重载的运算符是 ()。

- A. +
- B. =
- C. []
- D. ->

【7.4】在重载一个运算符为成员函数时,其参数表中没有任何参数,这说明该运算符是 ()。

- A. 后置单目运算符
- B. 前置单目运算符
- C. 无操作数的运算符
- D. 双目运算符

【7.5】C++中,重载的运算符“>>”是一个 ()。

- A. 用于输出操作的非成员函数
- B. 用于输入操作的非成员函数
- C. 用于输出操作的成员函数
- D. 用于输入操作的成员函数

【7.6】有如下程序:

```
#include<iostream>
using namespace std;
class Complex
{
    double re, im;
public:
    Complex(double r, double i):re(r), im(i){}
    double real()const{return re;}
    double image()const{return im;}
    Complex& operator+=(Complex a)
    {
        re+=a.re;
        im+=a.im;
        return *this;
    }
};
ostream& operator<<(ostream& s, const Complex& z)
{
    return s<<'<<z.real()<<', '<<z.image()<<';
}
int main()
{
    Complex x(1, -2), y(2, 3);
    cout<<(x+y)<<endl;
    return 0;
}
```

执行这个程序的输出结果是 ()。

- A. (1, -2)
- B. (2, 3)
- C. (3, 5)
- D. (3, 1)

【7.7】写出下列程序的运行结果。

```
#include<iostream>
using namespace std;
class D {
private:
    double d;
public:
    D()
```

```

    { d=0;
      cout<<"不带参数的构造函数"<<endl;
    }
    D(double i)
    { d=i;
      cout<<"转换构造函数"<<endl;
    }
    void print()
    { cout<<d<<endl; }
};

int main()
{ D d1;
  d1=D(12.3);
  d1.print();
  return 0;
}

```

【7.8】写出下列程序的运行结果。

```

#include<iostream>
using namespace std;
class T {
private:
    int a;
public:
    T(int a1)
    { a=a1; }
    operator double()
    { return double(a); }
};

int main()
{ T t(10);
  double s1=4.5, d;
  d=s1+t;
  cout<<d<<endl;
  return 0;
}

```

【7.9】编一个程序，用成员函数重载运算符“+”和“-”将两个二维数组相加和相减，要求第1个二维数组的值由构造函数设置，另一个二维数组的值由键盘输入。

【7.10】修改上题，用友元函数重载运算符“+”和“-”将两个二维数组相加和相减。

【7.11】为Date类重载“+”运算符，实现在某一个日期上（月、日、年）加一个天数。

Date类如下：

```

class Date{
public:
    Date() { }
    Date(int m, int d, int y)
    { month=m;
      day=d;
      year=y;
    }
    void print()
    { cout<<year<<". "<<month<<". "<<day<<endl; }
    Date operator +(int);
}

```

```
private:
    int month, day, year;
};
```

【7.12】定义一个矩阵类 *Matrix*，重载运算符“+”，使之能用于矩阵的加法运算。有两个矩阵 *a* 和 *b*，均为 2 行 4 列。求两个矩阵之和。重载流插入运算符“<<”和流提取运算符“>>”，使之能用于该矩阵的输入和输出。

第 8 章

模 板

模板是 C++ 的一个重要特性。模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现了代码的重用，使得一段程序可以用于处理多种不同类型的对象。利用模板机制可以显著减少冗余信息，能大幅度地节约程序代码，提高程序设计的效率，进一步提高面向对象程序的可重用性和可维护性。本章主要讲述模板的概念、函数模板与模板函数、类模板与模板类等内容。

8.1 模板的概念

C++ 允许用同一个函数名定义多个函数，这些函数的参数个数和参数类型不同。例如定义求最大值函数 `max` 时，需要对不同的数据类型分别定义不同的函数，例如：

```
int max(int a, int b)
{ return (a>b)? a:b;
}

long max(long a, long b)
{ return (a>b)? a:b;
}

double max(double a, double b)
{ return (a>b)? a:b;
}
```

虽然在上面的多个函数中，函数体都是一样，但是由于它们所处理的参数类型和返回值类型都不一样，所以是完全不同的函数。在 C++ 中，确实可以通过重载这些函数使它们拥有同样的函数名，但还是不得不为每个函数编写一组代码。如果能够使这些函数只写一遍，即写一个通用的函数，而它可以适用于多种不同的数据类型，便会使代码的可重用性大大提高，从而提高软件的开发效率。C++ 提供的模板就可以解决这个问题，模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现了真正的代码可重用性。使用模板可以大幅度地提高程序设计的效率。模板分为函数模板和类模板，它们分别允许用户构造模板函数和模板类。

8.2 函 数 模 板

8.2.1 函数模板的声明

函数模板实际上是建立一个通用函数，其函数返回值类型和形参类型不具体指定，用一个虚拟的类型来代表。这个通用函数就称为函数模板。凡是函数体相同的函数都可以用这个模板来替代，不必定义多个函数。在调用函数时系统会根据实参的类型（模板实参）来取代模板中虚拟类型从而实现了不同函数的功能。

函数模板的声明格式如下：

```
template<typename 类型参数>
返回类型 函数名(模板形参表)
```

```
{
    函数体
}
```

也可以定义成如下形式：

```
template<class 类型参数>
返回类型 函数名(模板形参表)
```

```
{
    函数体
}
```

其中，`template` 是一个声明模板的关键字，它表示声明一个模板。类型参数（通常用 C++ 标识符表示，如 `T`、`Type` 等）实际上是一个虚拟的类型名，使用前并未指定它是哪一种具体的类型，但使用函数模板时，必须将类型参数实例化。类型参数前需要加关键字 `typename`（或 `class`），`typename` 和 `class` 的作用相同，都是表示其后的参数是一个虚拟的类型名（即类型参数）。早期版本的 C++ 程序都用关键字 `class`，由于 `class` 容易与类名混淆，所以后来标准 C++ 又增加了关键字 `typename`，二者可以互换，但 `typename` 的含义比 `class` 清晰。

例如，将求最大值函数 `max` 定义成函数模板，如下所示：

```
template<typename T>                //T 为类型参数
T max(T a, T b)                    // "T a, T b" 为模板形参表
{ return (a>b)?a:b; }
```

也可以定义成如下形式：

```
template <class T>                  //T 为类型参数
T max(T a, T b)                    // "T a, T b" 为模板形参表
{ return (a>b)?a:b; }
```

8.2.2 函数模板的使用

上面定义的 `max` 函数代表的是一类函数，若要使用这个 `max` 函数进行求最大值操作，必

须将关键字 `typename` (或 `class`) 后面的类型参数 `T` 实例化为确定的数据类型 (如 `int` 等), 从这个意义上说, 它不是一个完全的函数, 我们称之为函数模板。将 `T` 实例化的参数称为模板实参, 用模板实参实例化的函数称为模板函数。

当编译系统发现有一个函数调用:

函数名(模板实参表);

时, 将根据模板实参表中的类型生成一个函数即模板函数。该模板函数的函数体与函数模板的函数体相同。

下面是使用上面定义的函数模板 `max` 的完整程序:

例 8.1 函数模板的使用。

```
#include<iostream>
using namespace std;
template <typename T>                //模板声明, 其中 T 为类型参数
T max(T a, T b)                      //定义函数模板, "T a, T b" 为模板形参表
{ return (a>b) ? a:b; }
int main()
{ int i1=10, i2=56;
  double d1=50.344, d2=4656.346;
  char c1='k', c2='n';
  cout<<"较大的整数是:"<<max(i1, i2)<<endl;
                                     //调用函数模板, i1 和 i2 为模板实参
  cout<<"较大的双精度型数是:"<<max(d1, d2)<<endl;
                                     //调用函数模板, 此时 T 被 double 取代
  cout<<"较大的字符是:"<<max(c1, c2)<<endl;
                                     //调用函数模板, 此时 T 被 char 取代
  return 0;
}
```

在此程序中, 用 "`max(i1, i2)`" 调用函数模板时, 用模板实参 `i1` 和 `i2` 的类型 `int` 取代函数模板中的类型参数 `T`, 此时相当于已定义了一个函数:

```
int max(int a, int b)
{ return (a>b) ? a:b;
}
```

然后调用它。用 "`max(d1, d2)`" 和 "`max(c1, c2)`" 调用函数模板的情况类似, 分别相当于已定义以下函数:

```
double max(double a, double b)
{ return (a>b) ? a:b;
}
```

和

```
char max(char a, char b)
{ return (a>b) ? a:b;
}
```


程序运行结果如下:

较大的整数是:56

较大的双精度型数是:4656.35

较大的字符是:n

从以上例子我们可以看出, 函数模板提供了一类函数的抽象, 它以类型参数 T 为函数参数及函数返回值的虚拟类型。函数模板经实例化而生成的具体函数称为模板函数。函数模板代表了一类函数, 模板函数表示某一具体的函数。图 8.1 给出了函数模板和模板函数的关系。

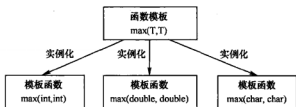


图 8.1 函数模板与模板函数之间的关系

说明:

(1) 在函数模板中允许使用多个类型参数。但是应当注意 `template` 定义部分的每个类型参数前必须有关键字 `typename` (或 `class`)。例如, 下面这个程序中建立了有两个类型参数的函数模板。

例 8.2 有两个类型参数的函数模板。

```
#include<iostream>
using namespace std;
template<typename para1, typename para2>           //模板声明, 有两个类型参数
void two_para(para1 x, para2 y)                   //定义函数模板
{ cout<<x<<' '<<y<<endl; }
int main()
{ two_para(99, "zhang");
  two_para(123.45, 888L);
  return 0;
}
```

程序运行结果如下:

99 zhang

123.45 888

在此程序中, 生成了两个模板函数, 其中 `two_para(99, "zhang")` 分别用模板实参 `int` 和 `char*` 将类型参数 `para1` 和 `para2` 进行了实例化。`two_para(123.45, 888L)` 分别用模板实参 `double` 和 `long` 将类型参数 `para1` 和 `para2` 进行了实例化。

(2) 在 `template` 语句与函数模板定义语句之间不允许插入别的语句。例如下面的程序段就不能编译。

```
template< typename T>
int i;           //错误, 在 template 语句与函数模板定义语句之间不允许插入别的语句
```

```
T min( T x, T y)
{ return ( x<y)?x:y; }
```

(3) 同一般函数一样, 函数模板也可以重载。

例 8.3 函数模板重载举例。

```
#include<iostream>
using namespace std;
template<typename Type>                //模板声明, 其中 Type 为类型参数
Type min(Type x, Type y)                //定义有两个类型参数的函数模板 min
{ return (x<y)?x:y; }
template <typename Type>
Type min(Type x, Type y, Type z)        //定义有 3 个类型参数的函数模板 min
{ Type t;
  t=(x<y)? x:y;
  return (t<z)?t:z;
}
int main()
{ int m=10, n=20, min2;
  double x=10.1, y=20.2, z=30.3, min3;
  min2=min(m, n);
  min3=min(x, y, z);
  cout<<"min{"<<m<<"", "<<n<<"")=<<min2<<endl;
                                     //调用有两个类型参数的模板函数 min
  cout<<"min{"<<x<<"", "<<y<<"", "<<z<<"")=<<min3<<endl;
                                     //调用有 3 个类型参数的模板函数 min

  return 0;
}
```

读者自己不难分析, 这个程序运行的结果为:

```
min(10, 20)=10
min(10.1, 20.2, 30.3)=10.1
```

(4) 函数模板与同名的非模板函数可以重载。在这种情况下, 调用的顺序是: 首先寻找一个参数完全匹配的非模板函数, 如果找到了就调用它; 若没有找到, 则寻找函数模板, 将其实例化, 产生一个匹配的模板函数, 若找到了, 就调用它。恰当运用这种机制, 可以很好地处理一般与特殊的关系。

例 8.4 函数模板与非模板函数重载举例。

```
#include<iostream>
using namespace std;
template<typename AT>                  //模板声明, 其中 AT 为类型参数
AT add(AT x, AT y)                    //定义函数模板, "AT x, AT y" 为模板形参表
{ cout<<"调用模板函数, ";
  return x+y;
}
int add(int x, int y)                  //定义非模板函数 add, 与函数模板 add 重载
{ cout<<"调用非模板函数, ";
  return x+y;
}
int main()
```

```

{ int i1=10, i2=56;
  double d1=50.34, d2=4656.34;
  cout<<"两个整数的和是:"<<add(i1, i2)<<endl;    //调用非模板函数
  cout<<"两个双精度型数的和是:"<<add(d1, d2)<<endl;
                                              //调用模板函数, 此时AT被double替代
return 0;
}

```

程序运行结果为:

调用非模板函数, 两个整数的和是:66

调用模板函数, 两个双精度型数的和是:4706.68

8.3 类 模 板

通过上一节的介绍, 可以看到: 使用函数模板能够简化程序的设计。对于类的声明来说, 也可以采用类似的方法, 通过使用类模板来简化那些功能相同而数据类型不同的类的声明。

所谓类模板, 实际上是建立一个通用类, 其数据成员、成员函数的返回类型和形参类型不具体指定, 用一个虚拟的类型来代表。使用类模板定义对象时, 系统会根据实参的类型来取代类模板中虚拟类型从而实现了不同类的功能。

定义一个类模板与定义函数模板的格式类似, 必须以关键字 `template` 开始, 后面是尖括号括起来的模板参数, 然后是类名, 其格式如下:

```

template <typename 类型参数>
class 类名 {
    类成员声明
};

```

也可以定义成如下形式:

```

template <class 类型参数>
class 类名 {
    类成员声明
};

```

与函数模板类似, 其中, `template` 是一个声明模板的关键字, 它表示声明一个模板。类型参数 (通常用 C++ 标识符表示, 如 `T`、`Type` 等) 实际上是一个虚拟的类型名, 现在并未指定它是哪一种具体的类型, 但使用类模板时, 必须将类型参数实例化。类型参数前需要加关键字 `typename` (或 `class`), `typename` 和 `class` 的作用相同, 都是表示其后的参数是一个虚拟的类型名 (即类型参数)。

在类声明中, 欲采用通用数据类型的数据成员、成员函数的参数或返回类型前面需加上类型参数。

例如, 建立一个用来实现求 3 个数之和的类模板。

```

template<typename T>                //模板声明, 其中T为类型参数
class Add_3{                        //类模板名为Add_3

```

```

public:
    Add_3(T a, T b, T c)
    { x=a; y=b; z=c; }
    T sum()
    { return x+y+z; }
private:
    T x, y, z;
};

```

用类模板定义对象时，采用以下形式：

类模板名<实际类型名> 对象名；

类模板名<实际类型名> 对象名(实参表列)；

因此，使用上面求 3 个数之和的类模板的主函数可写成：

```

int main()
{ Add_3 <int> sum3_1(3, 5, 7);
  Add_3 <double> sum3_2(12.34, 34.56, 56.78);
  cout<<"三个数之和是："<< sum3_1.sum()<<endl;
  cout<<"三个数之和值是："<< sum3_2.sum()<<endl;
  return 0;
}

```

例 8.5 类模板的使用举例。

```

#include<iostream>
using namespace std;
template<typename T>           // 模板声明，其中 T 为类型参数
class Add_3{                   // 类模板名为 Add_3
public:
    Add_3(T a, T b, T c)
    { x=a; y=b; z=c; }
    T sum()
    { return x+y+z; }
private:
    T x, y, z;
};

int main()
{ Add_3 <int> sum3_1(3, 5, 7);           // 用类模板定义对象 sum3_1，此时 T 被 int 替代
  Add_3 <double> sum3_2(12.34, 34.56, 56.78);
                                           // 用类模板定义对象 sum3_2，此时 T 被 double 替代
  cout<<"三个整数之和是："<< sum3_1.sum()<<endl;
  cout<<"三个双精度数之和是："<< sum3_2.sum()<<endl;
  return 0;
}

```

程序运行结果如下：

三个整数之和是：15

三个双精度数之和是：103.68

在以上例子中，成员函数（其中含有类型参数）是定义在类体内的。但是，类模板中的

成员函数也可以在类模板体外定义。此时，若成员函数中有类型参数存在，则 C++ 有一些特殊的规定：

- (1) 需要在成员函数定义之前进行模板声明；
- (2) 在成员函数名前缀上“类名<类型参数>::”。

在类模板体外定义的成员函数的一般形式如下：

```
template <typename 类型参数>
函数类型 类名<类型参数>::成员函数名(形参表)
{
    成员函数体
}
```

例如，上例中成员函数 sum 在类模板体外定义时，应该写成：

```
template<typename T>
T Add_3<T>::sum()
{ return x+y+z;
}
```

下面是成员函数 sum 定义在类模板体外时的完整例子。

例8.6 在类模板体外定义成员函数举例。

```
#include<iostream>
using namespace std;
template<typename T>                //模板声明，其中 T 为类型参数
class Add_3{                        //类模板名为 Compare
public:
    Add_3(T a, T b, T c);          //声明构造函数的原型
    T sum();                        //声明成员函数 max 的原型
private:
    T x, y, z;
};
template<typename T>                //模板声明
Add_3<T>::Add_3(T a, T b, T c)      //在类模板体外定义构造函数
{ x=a; y=b; z=c; }
template<typename T>                //模板声明
T Add_3<T>::sum()                  //在类模板体外定义成员函数 sum，返回类型为 T
{ return x+y+z; }
int main()
{ Add_3 <int> sum3_1(3, 5, 7);      //用类模板定义对象 sum3_1，此时类型参数 T 被 int 替代
    Add_3 <double> sum3_2(12.34, 34.56, 56.78);
    //用类模板定义对象 sum3_2，此时类型参数 T 被 double 替代
    cout<<"三个整数之和是："<< sum3_1.sum()<<endl;
    cout<<"三个双精度数之和是："<< sum3_2.sum()<<endl;
    return 0;
}
```

程序运行结果如下：

三个整数之和是：15

三个双精度数之和是：103.68

此例中，类模板 `Add_3` 经实例化后生成了两个类型分别为 `int` 和 `double` 的模板类，这两个模板类经实例化后又生成了两个对象 `sum3_1` 和 `sum3_2`。类模板代表了一类类，模板类表示某一具体的类。图 8.2 给出了类模板、模板类和对象之间的关系。

下面再看一个程序，在这个程序中建立了一个用来实现堆栈的类模板 `Stack`。

例 8.7 类模板 `Stack` 的使用举例。

在此例子中建立了字符型和整型两个堆栈。

```
#include<iostream>
using namespace std;
const int size=10;
template< typename Type>           //模板声明，其中 Type 为类型参数
class Stack{                       //类模板名为 Stack
public:
    void init()
    { tos=0; }
    void push(Type ch);             //声明成员函数 push 的原型， 函数参数为 Type 类型
    Type pop();                     //声明成员函数 pop 的原型，返回类型为 Type 类型
private:
    Type stck[size];               //数组类型为 Type， 即数组可取任意类型
    int tos;
};
template<typename Type>            //模板声明
void Stack<Type>::push(Type ob)    //在类模板体外定义成员函数 push
{ if (tos==size)
    { cout<<"Stack is full";
      return ;
    }
    stck [tos]=ob;
    tos++;
}
template <typename Type >         //模板声明
Type Stack <Type>::pop()           //在类模板体外定义成员函数 pop
{ if (tos==0)
    { cout<<"Stack is empty";
      return 0;
    }
    tos--;
    return stck[tos];
}
int main()
{
    Stack <char>s;                 //定义字符堆栈
                                   //用类模板定义对象 s，此时 Type 被 char 替代
```

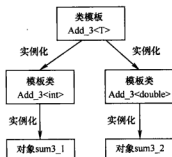


图 8.2 类模板、模板类和对象之间的关系

```

int i;
s.init();
s.push('a');
s.push('b');
s.push('c');
for(i=0;i<3;i++) cout<<"pop s: "<<s.pop()<<endl;
                                //定义整型堆栈

Stack <int> is;                    //用类模板定义对象 is, 此时 Type 被 int 替代
is.init();
is.push(1);
is.push(3);
is.push(5);
for (i=0;i<3;i++)
    cout<<"pop is: "<<is.pop()<<endl;
return 0;
}

```

程序运行结果如下:

```

pop s: c
pop s: b
pop s: a
pop is: 5
pop is: 3
pop is: 1

```

在上面的程序中建立了一个用来实现堆栈的类模板。

```

template< typename Type>          //模板声明, 其中 Type 为类型参数
class Stack{                      //类模板名为 Stack
public:
    void init()
    { tos=0;
    }

    void push(Type ch);            //声明成员函数 push 的原型, 函数参数为 Type 类型
    Type pop();                   //声明成员函数 pop 的原型, 返回类型为 Type 类型
private:
    Type stck[size];              //数组类型为 Type, 即数组可取任意类型
    int tos;
};

```

在类模板外定义成员函数 push 和 pop 时, 由于成员函数中有类型参数存在, 则需要在函数外进行模板声明, 并且在函数名前缀上“类名<类型参数>::”。成员函数 push() 和 pop() 在类模板外定义为:

```

template<typename Type>          //模板声明
void Stack<Type>::push(Type ob)  //在类模板体外定义成员函数 push
{ if (tos==size)
    { cout<<"Stack is full";
      return ;
    }
    stck [tos]=ob;
    tos++;
}

```

```

}
template <typename Type>           //模板声明
Type Stack <Type>::pop()           //在类模板体外定义成员函数 pop
{ if (tos==0)
    { cout<<"Stack is empty";
      return 0;
    }
    tos--;
    return stk[tos];
}

```

在函数 main 中,用语句“Stack <char>s;”建立了 char 型的对象 s,用语句“Stack <int> is;”建立了 int 型的对象 is。在 main 函数中我们还可以定义其他类型的类对象,例如可以用以下语句建立 double 型对象 ds:

```
Stack <double> ds;
```

说明:

在每个类模板定义之前,都需要在前面加上模板声明,如例 8.7 中需加上:

```
template <typename Type>
```

或

```
template <class Type>
```

类模板在使用时,必须在类模板名字后面缀上<类型参数>,如例 8.7 中需加上:

```
Stack <Type>
```

模板类可以有多个类型参数,在下面的短例中建立了使用两个类型参数的类模板。

例 8.8 有两个类型参数的类模板举例。

```

#include<iostream>
using namespace std;
template<typename T1, typename T2>           //声明具有两个类型参数的模板
class MyClass{                               //定义类模板 MyClass
public:
    MyClass(T1 a, T2 b)
    { i=a; j=b;}
    void show()
    { cout<<"i="<<i<<" j="<<j<<endl;}
private:
    T1 i;
    T2 j;
};
int main()
{ MyClass <int, double> ob1(12, 0.15);
    //用类模板定义对象 ob1, 此时 T1、T2 分别被 int 与 double 取代
    MyClass <char, char *> ob2('x', "This is a test.");
    //用类模板定义对象 ob2, 此时 T1、T2 分别被 char 与 char* 取代
    ob1.show();
    ob2.show();
    return 0;
}

```


程序运行结果如下:

```
i=12 j=0.15
i=x j=This is a test.
```

这个程序声明了一个类模板,它具有两个类型参数。在 main 函数中定义了两种类型的对象, ob1 使用了 int 型与 double 型数据, ob2 使用了 char 型和 char* 型数据。

8.4 应用举例

例 8.9 编写一个使用类模板对数组进行排序、查找和求元素和的程序,并采用相关数据进行测试。

```
#include<iostream.h>
#include<iomanip.h>
template<class Type>
class Array {                                //声明类模板 Array
    Type* set;
    int n;
public:
    Array(Type* data, int i)
    { set=data;
      n=i;
    }
    ~Array() {}
    void sort();                             //成员函数,用于对数组进行排序
    int seek(Type key);                      //成员函数,用于查找指定的元素
    Type sum();                             //成员函数,用于求数组元素之和
    void disp();                             //成员函数,用于显示所有元素
};

template<class Type>
void Array<Type>::sort()                    //定义成员函数,采用冒泡排序法排序
{ Type temp;
  for(int i=1;i<n;i++)
    for(int j=n-1;j>=i;j--)
      if(set[j-1]>set[j])
      { temp=set[j-1];
        set[j-1]=set[j];
        set[j]=temp;
      }
}

template<class Type>
int Array<Type>::seek(Type key)             //定义成员函数,采用顺序查找法查找元素
{ for(int i=1;i<n+1;i++)
  { if(set[i-1]==key)
    { cout<<"    "<<key<<"是数组中的第"<<i<<"个元素"<<endl;
      break;
    }
    if(set[i-1]!=key)
      cout<<"    "<<key<<"不是数组中的元素!"<<endl;
  }
  return 0;
}
```

```

template<class Type>
Type Array<Type>::sum()           //定义成员函数，用于求数组元素之和
{
    Type s=0;
    for(int i=0;i<n;i++)
        s+=set[i];
    cout<<"    数组元素之和为:"<<s<<endl;
    return 0;
}

template<class Type>
void Array<Type>::disp()         //定义成员函数，输出数组元素
{
    for(int i=0;i<n;i++)
        cout<<set[i]<<" ";
    cout<<endl;
}

int main()
{
    int a[]={66, 33, 88, 11, 99, 44, 77, 55, 22};
    double b[]={5.5, 2.2, 6.6, 3.3, 8.8, 1.1};

    Array<int> arr1(a, 9);
    Array<double> arr2(b, 6);

    cout<<"数组 arr1:"<<endl;
    cout<<"    原数组序列为:";
    arr1.disp();
    arr1.sum();
    arr1.seek(11);
    arr1.sort();
    cout<<"    排序后数组序列为:";
    arr1.disp();
    cout<<"数组 arr2:"<<endl;
    cout<<"    原数组序列为:";
    arr2.disp();
    arr2.sum();
    arr2.seek(6.4);
    arr2.sort();
    cout<<"    排序后数组序列为:";
    arr2.disp();
}

```

程序运行结果如下：

数组 arr1:

原数组序列为:66 33 88 11 99 44 77 55 22

数组元素之和为:495

11 是数组中的第 4 个元素

排序后数组序列为:11 22 33 44 55 66 77 88 99

数组 arr2:

原数组序列为:5.5 2.2 6.6 3.3 8.8 1.1

数组元素之和为:27.5

6.4 不是数组中的元素!

排序后数组序列为:1.1 2.2 3.3 5.5 6.6 8.8

实 验

实验目的和要求

1. 正确理解模板的概念。
2. 学习函数模板和类模板的声明和使用方法。

实验内容和步骤

1. 分析并调试下列程序，写出运行结果并分析原因。

(1)

```
//test8_1_1.cpp
#include<iostream>
using namespace std;
template<typename T>
T max(T x, T y)                                //函数模板
{ return x>y?x:y; }
int max(int a, int b)                          //非模板函数
{ return a>b?a:b; }
double max(double a, double b)                //非模板函数
{ return a>b?a:b; }
int main()
{ cout<<"max('3', '7') is "<<max('3', '7')<<endl;
  return 0;
}
```

(2)

```
//test8_1_2.cpp
#include<iostream>
using namespace std;
int max(int a, int b)                          //非模板函数
{ return a>b?a:b; }
double max(double a, double b)                //非模板函数
{ return a>b?a:b; }
int main()
{ cout<<"max('3', '7') is "<<max('3', '7')<<endl;
  return 0;
}
```

2. 编写一个求任意类型数组中最大元素和最小元素的程序，要求将求最大元素和最小元素的函数设计成函数模板。

3. 编写一个程序，使用类模板对数组元素进行排序、倒置、查找和求和。

【提示】

设计一个类模板：

```
template<class Type>
class Array{
...
};
```

具有对数组元素进行排序、倒置、查找和求和的功能，然后用它产生类型实参分别为 `int` 型和 `double` 型的两个模板类，分别对整型数组与双精度数组完成所要求的操作。

习 题

【8.1】什么是函数模板？函数模板声明的一般形式是什么？

【8.2】函数模板与同名的非模板函数重载时，调用的顺序是怎样的？

【8.3】什么是类模板？类模板声明的一般形式是什么？

【8.4】使用模板是为了（ ）。

- A. 提高代码的可重用性 B. 提高代码的运行效率
C. 加强类的封装性 D. 实现多态性

【8.5】关于类模板，下列表述不正确的是（ ）。

- A. 用类模板定义一个对象时，不能省略实参
B. 类模板只能有一个类型参数
C. 在类模板定义之前，都需要在前面加上模板声明
D. 类模板只能有虚拟类型参数

【8.6】类模板的使用实际上是将类模板实例化成一个具体的（ ）。

- A. 函数 B. 对象 C. 类 D. 模板类

【8.7】（ ）允许用户为类定义一种模式，使得类中的某些数据成员、某些成员函数的参数和返回值能取任意数据类型。

- A. 类模板 B. 模板函数 C. 函数模板 D. 模板类

【8.8】类模板的模板参数（ ）。

- A. 只可作为成员函数的参数类型
B. 只可作为成员函数的返回类型
C. 只可作为数据成员的类型
D. 以上三者皆可

【8.9】以下对模板的说明，正确的是（ ）。

- A. `template<class T1, class T2>` B. `template<class T1, T2>`
C. `template<T>` D. `template<class T; class T2>`

【8.10】写出下面程序的运行结果。

```
#include<iostream>
using namespace std;
template<class T>
T min(T a, T b)
{ if(a<b) return a;
  else return b;
}
int main()
{ int n1=9, n2=6;
  double d1=0.5, d2=4.8;
  cout<<"较小整数:"<<min(n1, n2)<<endl;
```

```

    cout<<"较小实数:"<<min(d1, d2)<<endl;
    return 0;
}

```

【8.11】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
template <class Type1, class Type2>
class myclass{
public:
    myclass(Type1 a, Type2 b)
    { i=a; j=b; }
    void show()
    { cout<<i<<' '<<j<<'\n'; }
private:
    Type1 i;
    Type2 j;
};

int main()
{ myclass<int, double> ob1(10, 0.23);
  myclass<char, char*> ob2('X', "This is a test.");
  ob1.show();
  ob2.show();
  return 0;
}

```

【8.12】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
class coord
{ int x, y;
public:
    coord(int x1, int y1){x=x1;y=y1;}
    int getx(){return x;}
    int gety(){return y;}
    int operator<(coord& c);
};

int coord::operator<(coord& c)
{ if(x<c.x)
  if(y<c.y)
    return 1;
  return 0;
}

template<class obj>
obj& min(obj& o1, obj& o2)
{ if(o1<o2)
  return o1;
  return o2;
}

int main()
{ coord c1(5, 12);
  coord c2(3, 16);
}

```

```

coord c3=min(c1, c2);
cout<<"最小的坐标: ("<<c3.getx()<<" "<<c3.gety()<<" "<<endl;
double d1=2.99;
double d2=3.48;
cout<<"最小的数:"<<min(d1, d2)<<endl;
return 0;
}

```

【8.13】指出下列程序中的错误，并说明原因。

```

#include<iostream>
using namespace std;
template <typename T>                //模板声明，其中为T类型参数
class Compare{                       //类模板名为 Compare
public:
    Compare(T a, T b)
    { x=a; y=b;}
    T min();
private:
    T x, y;
};
template <typename T>
T Compare::min()
{ return (x<y)? x:y; }
int main()
{ Compare com1(3, 7);
  cout<<"其中的最小值是:"<< com1.min()<<endl;
  return 0;
}

```

【8.14】已知下列主函数：

```

int main()
{ cout<<min(10, 5, 3)<<endl;
  cout<<min(10.0, 5.0, 3.0)<<endl;
  cout<<min('a', 'b', 'c')<<endl;
  return 0;
}

```

设计一个求3个数中最小者的函数模板，并写出调用此函数模板的完整程序。

【8.15】编写一个函数模板，求数组中的最大元素，并写出调用此函数模板的完整程序，使得函数调用时，数组的类型可以是整数也可以是双精度类型。

【8.16】编写一个函数模板，使用冒泡排序将数组内容由小到大排列并打印出来，并写出调用此函数模板的完整程序，使得函数调用时，数组的类型可以是整数也可以是双精度型。

【8.17】建立一个用来实现求3个数和的类模板（将成员函数定义在类模板的内部），并写出调用此类模板的完整程序。

【8.18】将8.17题改写为在类模板外定义各成员函数。

第9章

C++的输入和输出

C++系统提供了一个用于输入输出(I/O)操作的类体系,这个类体系提供了对预定义数据类型进行输入输出操作的能力,程序员也可以利用这个类体系对自定义数据类型进行输入输出操作。

9.1 C++流的概述

9.1.1 C++的输入/输出流

在C++中,输入输出是通过流来完成的。流指的是数据从一个源流到一个目的的抽象,它负责在数据的生产者(源)和数据的消费者(目的)之间建立联系,并管理数据的流动。凡是数据从一个地方传输到另一个地方的操作都是流的操作,从流中提取数据称为输入(通常又称为提取)操作,向流中添加数据称为输出(通常又称为插入)操作。

C++的输入输出是以字节流的形式实现的。在输入操作中,字节流从输入设备(例如键盘、磁盘、网络连接等)流向内存;在输出操作中,字节流从内存流向输出设备(例如显示器、打印机、网络连接等)。字节流可以是ASCII字符、二进制形式的数据、图形图像、音频视频等信息。文件和字符串也可以看成有序的字节流,分别称为文件流和字符串流。

与C一样,C++中也没有输入输出语句。C++编译系统带有一个面向对象的输入输出软件包,它就是C++的I/O流类库。在I/O流类库中包含许多用于输入输出的类,称为流类。用流类定义的对象称为流对象。

1. 用于输入输出的头文件

C++编译系统带有一个面向对象的输入输出软件包,它就是I/O流类库。I/O流类库提供了数百种输入输出功能,I/O流类库中各种的类的声明被放在相应的头文件中,用户在自己的程序中用#include命令包含了有关的头文件就相当于在本程序中声明了所需要用到的类。常用的头文件有:

- **iostream** 包含了对输入输出流进行操作所需的基本信息。使用cin、cout等流对象进行针对标准设备的I/O操作时,须包含此头文件。
- **fstream** 用于用户管理文件的I/O操作。使用文件流对象进行针对磁盘文件的操作,须包含此头文件。
- **stringstream** 用于字符串流的I/O操作。使用字符串流对象进行针对内存字符串空间的

I/O 操作, 须包含此头文件。

- `iomanip` 用于输入输出的格式控制。在使用 `setw`、`fixed` 等大多数操作符进行格式控制时, 须包含此头文件。

2. 用于输入输出的流类

I/O 流类库中包含了许多用于输入输出操作的类, 其中类 `istream` 支持流输入操作, 类 `ostream` 支持流输出操作, 类 `iostream` 同时支持流输入和输出操作。表 9.1 表列出了 I/O 流类库中常用的流类, 以及指出了这些流类在哪个头文件中声明。

表 9.1 I/O 流类库中的常用流类

类 名	说 明	头文件
抽象流基类		
<code>ios</code>	流基类	<code>iostream</code>
输入流类		
<code>istream</code>	通用输入流类和其他输入流的基类	<code>istream</code>
<code>ifstream</code>	输入文件流类	<code>fstream</code>
<code>istrstream</code>	输入字符串流类	<code>strstream</code>
输出流类		
<code>ostream</code>	通用输出流类和其他输出流的基类	<code>iostream</code>
<code>ofstream</code>	输出文件流类	<code>fstream</code>
<code>ostrstream</code>	输出字符串流类	<code>strstream</code>
输入输出流类		
<code>iostream</code>	通用输入输出流类和其他输入输出流的基类	<code>iostream</code>
<code>fstream</code>	输入输出文件流类	<code>fstream</code>
<code>strstream</code>	输入输出字符串流类	<code>strstream</code>

`ios` 是抽象基类, 类 `istream` 和 `ostream` 是通过单继承从基类 `ios` 派生而来的, 类 `iostream` 是通过多重继承从类 `istream` 和 `ostream` 派生而来的, 继承的层次结构如图 9.1 所示。

`ios` 作为流类库中的一个基类, 还可以派生出许多类, 其中有四个直接派生类, 即输入流类 (`istream`)、输出流类 (`ostream`)、文件流类 (`fstreambase`) 和串流类 (`strstreambase`), 这四种流作为流库中的基本流类。

以 `istream`、`ostream`、`fstreambase` 和 `strstreambase` 四个基本流类为基础还可以派生出多个实用的流类, 例如: `ifstream` (输入文件流类)、`ofstream` (输出文件流类)、`fstream` (输入输出文件流类)、`istrstream` (输入字符串流类)、`ostrstream` (输出字符串流类) 和 `strstream` (输入输出字符串流类) 等。

9.1.2 预定义的流对象

用流类定义的对象称为流对象。与输入设备 (如键盘) 相联系的流对象称为输入流对象;



图 9.1 输入输出流类的继承层次结构

与输出设备（如屏幕）相联系的流对象称为输出流对象。

C++中包含几个预定义的流对象，它们是标准输入流对象 `cin`、标准输出流对象 `cout`、非缓冲型的标准出错流对象 `cerr` 和缓冲型的标准出错流对象 `clog`。

`cin` 是 `istream` 类的对象，它与标准输入设备（通常指键盘）相联系，用于进行输入操作。

`cout` 是 `ostream` 类的对象，它与标准输出设备（通常指显示器）相联系，用于进行输出操作。

`cerr` 是 `ostream` 类的对象，它与标准错误输出设备（通常指显示器）相联系，用于输出出错信息。

`clog` 是 `ostream` 类的对象，它与标准错误输出设备（通常指显示器）相联系，用于输出出错信息。

`cin` 与 `cout` 的使用方法，在前面的章节中我们已经作了介绍。`cerr` 与 `clog` 均用来输出出错信息。`cerr` 和 `clog` 之间的区别是，`cerr` 是不经过缓冲区，直接向显示器上输出有关信息，因而发送给它的任何内容都立即输出；相反，`clog` 中的信息存放在缓冲区中，缓冲区满后或遇上 `endl` 时向显示器输出。

由于 `istream` 和 `ostream` 类都是在头文件 `iostream` 中声明的，因此只要在程序中包含头文件 `iostream.h`，C++程序开始运行时这四个标准流对象的构造函数都被自动调用。

9.1.3 输入输出流的成员函数

在 C++ 程序中除了用 `cout` 和插入运算符 “<<” 实现输出，用 `cin` 和提取运算符 “>>” 实现输入外，还可以用类 `istream` 和类 `ostream` 流对象的一些成员函数，实现字符的输出和输入。下面介绍其中的一部分。

1. put 函数

`put` 函数用于输出一个字符，其常用的调用形式为：

```
cout.put(单字符);
```

或

```
cout.put(字符型变量);
```

例如，语句

```
cout.put('A');
```

将字符 A 显示在屏幕上，它与语句

```
cout<<'A';
```

等价，所不同的是 `put` 函数的参数不但可以是字符，还可以是字符的 ASCII 代码（也可以是一个整型表达式）。例如语句

```
cout.put(65);
```

或

```
cout.put(20+45);
```

都可以将字符 A 显示在屏幕上。

可以在一个语句中连续调用 `put` 函数，例如：

```
cout.put(65), cout.put(66), cout.put(67), cout.put('\n');
```

2. get 函数

get 函数的功能与提取运算符“>>”类似，主要的不同之处是 get 函数在读入数据时可包括空白字符，而提取运算符“>>”在默认情况下拒绝接收空白字符。

其常用的调用形式为：

cin.get(字符型变量)

其作用是从输入流中读取一个字符（包括空白符），赋给字符变量 ch，如果读取成功则函数返回非 0 值，如失败（遇文件结束符 EOF）则函数返回 0 值。

例 9.1 get 函数应用举例。

```
#include<iostream>
using namespace std;
int main()
{ char ch;
  cout<<"Input:";
  while(cin.get(ch))
    cout.put(ch);
  return 0;
}
```

运行时，如果输入：

123 abc xyz

则输出：

123 abc xyz

当输入“Ctrl+z”及回车时，程序读入的值是 EOF，程序结束。

3. getline 函数

getline 函数常用的调用形式为：

cin.getline(字符数组，字符个数 n[, 终止标志字符])

或

cin.getline(字符指针，字符个数 n[, 终止标志字符])

其功能是从输入流读取 n-1 个字符，赋给指定的字符数组(或字符指针指向的数组)，然后插入一个字符串结束标志'\n'。如果在读取 n-1 个字符之前遇到指定的终止字符，则提前结束读取，然后插入一个字符串结束标志'\n'。

例 9.2 用 getline 函数读入一行字符。

本程序连续读入一串字符，最多读取 19 个字符赋给字符数组 line，或遇到字符't'提前停止。

```
#include<iostream>
using namespace std;
int main()
{ char line[20];
  cout<<"输入一行字符:"<<endl;
  cin.getline(line, 20, 't'); //读入 19 个字符或遇字符't'结束
  cout<<line;
  return 0;
}
```

说明:

请注意用“cin>>”和成员函数“cin.getline()”读取数据有以下区别:

(1) 使用“cin>>”可以读取 C++ 标准类型的各类数据 (如果经过重载, 还可以用于输入自定义类型的数据), 而用“cin.getline()”只能用于输入字符型数据。

(2) 使用“cin>>”读取数据时以空白字符 (包括空格、tab 键、回车键) 作为终止标志, 而“cin.getline()”可连续读取一系列字符, 可以包括空格。

4. ignore 函数

ignore 函数常用的的调用形式为:

cin.ignore(n, 终止字符)

ignore 函数的功能是跳过输入流中 n 个字符 (默认个数为 1), 或在遇到指定的终止字符 (默认终止字符是 EOF) 时提前结束。如

```
cin.ignore(10, 't') //跳过流入流中 10 个字符, 或遇字符't'后就不再跳了
```

ignore 函数可以不带参数或只带一个参数, 如

```
cin.ignore() //只跳过 1 个字符 (n 的默认值为 1, 默认终止字符是 EOF)
```

相当于

```
cin.ignore(1, EOF)
```

9.2 预定义类型输入输出的格式控制

在很多情况下, 需要对预定义类型 (如 int、float、double 型等) 的数据的输入输出格式进行控制。在 C++ 中, 仍然可以使用 C 中的 printf 和 scanf 函数进行格式化。除此以外, C++ 还提供了两种进行格式控制的方法: 一种是使用 ios 类中有关格式控制的流成员函数进行格式控制; 另一种是使用称为操纵符的特殊类型的函数进行格式控制。下面介绍这两种格式控制的方法。

9.2.1 用流成员函数进行输入输出格式控制

ios 类中有几个流成员函数可以用来对输入输出进行格式控制。常用的流成员函数如表 9.2 所示。

表 9.2 用于控制输入输出格式的流成员函数

流成员函数	功 能
setf(flags)	设置状态标志, 待设置的状态标志 flags 的内容见表 9.3 所示
unsetf(flags)	清除状态标志, 待清除的状态标志 flags 的内容见表 9.3 所示
width(n)	设置字段域宽为 n 位
fill(char ch);	设置填充字符 ch
precision(n)	设置实数的精度为 n 位, 在以普通十进制小数形式输出时 n 代表有效数字。在以 fixed(固定小数位数)形式和 scientific(指数)形式输出时 n 为小数位数

流成员函数 setf 和 unsetf 括号中的参数是用状态标志指定的, 状态标志在类 ios 中被定义成枚举值, 所以在引用这些状态标志时要在前面加上“ios::”, 状态标志如表 9.3 所示。

表 9.3

类 ios 中定义的状态标志

状态标志	功 能	输入/输出
ios::skipws	跳过输入中的空白符	用于输入
ios::left	输出数据在本域宽范围内左对齐	用于输出
ios::right	输出数据在本域宽范围内右对齐	用于输出
ios::internal	数据的符号位左对齐, 数据本身右对齐, 符号和数据之间为填充符	用于输出
ios::dec	设置整数的基数为 10	用于输入/输出
ios::oct	设置整数的基数为 8	用于输入/输出
ios::hex	设置整数的基数为 16	用于输入/输出
ios::showbase	输出整数时显示基数符号(八进制数以 0 打头, 十六进制数以 0x 打头)	用于输入/输出
ios::showpoint	浮点数输出时带有小数点	用于输出
ios::uppercase	在以科学表示法格式 E 和以十六进制输出字母时用大写表示	用于输出
ios::showpos	正整数前显示“+”符号	用于输出
ios::scientific	用科学表示法格式(指数)显示浮点数	用于输出
ios::fixed	用定点格式(固定小数位数)显示浮点数	用于输出
ios::unitbuf	完成输出操作后立即刷新所有的流	用于输出
ios::stdio	完成输出操作后刷新 stdout 和 stderr	用于输出

下面分别介绍这些成员函数的使用方法。

1. 设置状态标志流成员函数 setf

设置状态标志, 即是将某一状态标志位置“1”, 可使用 setf 函数, 其一般格式为:

```
long ios::setf(long flags)
```

使用时, 其一般的调用格式为:

流对象.setf(ios::状态标志);

例如:

```
istream isobj;
ostream osobj;
isobj.setf(ios::skipws);           //跳过输入中的空白符
osobj.setf(ios::left);             //输出数据在本域宽范围内左对齐
```

在此, isobj 为类 istream 的流对象, osobj 为类 ostream 的流对象。实际上, 在编程中用得最多的是 cin.setf(...)或 cout.setf(...)。

例 9.3 设置状态标志举例。

```
#include<iostream>
using namespace std;
int main()
{ cout.setf(ios::showpos|ios::scientific);
  cout <<567<<" "<<567.89<<endl;
  return 0;
}
```

设置 showpos 使得每个正数前添加 “+” 号, 设置 scientific 使浮点数按科学表示法(指数形式)进行显示。

程序运行的结果为：

```
+567 +5.678900e+002
```

分析以上程序和运行结果。

(1) 由于状态标志在类 `ios` 中被定义成枚举值，所以在引用这些状态标志时要在前面加上 `"ios::"`。

(2) 在使用 `setf` 函数设置多项标志时，中间应该用或运算符 `"|"` 分隔，例如：

```
cout.setf(ios::showpos|ios::dec|ios::scientific);
```

2. 清除状态标志流成员函数 `unsetf`

清除某一状态标志，即是将某一状态标志位置“0”，可使用 `unsetf` 函数，它的一般格式为：

```
long ios::unsetf(long flags)
```

使用时的调用格式为：

```
流对象.unsetf(ios::状态标志);
```

流成员函数 `unsetf` 括号中的参数 `flags` 与流成员函数 `setf` 相同。

3. 设置域宽流成员函数 `width`

设置域宽的成员函数是 `width` 函数，其常用的格式为：

```
int ios::width(int n);
```

此函数用来设置域宽为 `n` 位。

使用时的调用格式为：

```
流对象.width(n);
```

注意，所设置的域宽仅对下一个流输出操作有效，当一次输出操作完成之后，域宽又恢复为默认域宽 0。

4. 设置实数的精度流成员函数 `precision`

设置显示精度的成员函数的一般格式为：

```
int ios::precision(int n);
```

设置实数的精度为 `n` 位，在以一般十进制小数形式输出时 `n` 代表有效数字。在以 `fixed`（固定小数位数）形式和 `scientific`（指数）形式输出时 `n` 为小数位数。

使用时的调用格式为：

```
流对象.precision(n);
```

5. 填充字符流成员函数 `fill`

填充字符的作用是：当输出值不满域宽时用填充字符来填充，默认情况下填充字符为空格。所以在使用填充字符函数 `fill` 时，必须与 `width` 函数相配合，否则就没有意义。填充字符的成员函数一般的格式为：

```
char ios::fill(char ch);
```

`ch` 为所要填充的字符。

使用时的调用格式为：

```
流对象.fill(ch);
```

例 9.4 在数据符号和数据之间插入指定的填充符。

```
#include<iostream>
```

```
using namespace std;
int main()
{ double i=-5.1;
  cout.width(10);
  cout.fill('*');
  cout.setf(ios::internal);
  cout<<i<<endl;
  return 0;
}
```

程序运行结果如下:

```
*****5.1
```

下面举一个例子来说明以上这些函数的作用。

例 9.5 流成员函数使用方法举例。

```
#include<iostream>
using namespace std;
int main()
{ cout<<"-----1-----\n";
  cout.width(10); //设置域宽为 10 位
  cout<<123<<endl; //输出整数 123, 占 10 位, 默认为右对齐
  cout<<"-----2-----\n";
  cout<<123<<endl; //输出整数 123, 上面的 width(10) 已不起作用
  //此时按系统默认的域宽输出 (按数据实际长度输出)

  cout<<"-----3-----\n";
  cout.fill('&'); //设置填充字符为 '&'
  cout.width(10); //设置域宽为 10 位
  cout<<123<<endl; //输出整数 123, 占 10 位, 默认为右对齐, 填充字符为 '&'
  cout<<"-----4-----\n";
  cout.setf(ios::left); //设置左对齐
  cout<<123<<endl; //输出整数 123, 上面的 width(10) 已不起作用,
  //按数据实际长度输出, 左对齐

  cout<<"-----5-----\n";
  cout.precision(4); //设置实数的精度为 4 位
  cout<<123.45678<<endl; //以一般十进制小数形式输出时, 有效数字为 4
  cout<<"-----6-----\n";
  cout.setf(ios::fixed); //用定点格式 (固定小数位数) 显示浮点数
  cout<<123.45678<<endl; //以 fixed 形式输出时, 小数位数占 4 位
  cout<<"-----7-----\n";
  cout.width(15); //设置域宽为 15 位
  cout.unsetf(ios::fixed); //清除用定点格式 (小数形式) 显示浮点数
  cout.setf(ios::scientific); //用科学表示法格式 (指数) 显示浮点数
  cout<<123.45678<<endl; //用科学表示法格式 (指数) 输出, 小数占 4 位
  cout<<"-----8-----\n";
  int a=21;
  cout.setf(ios::showbase); //输出整数时显示基字符号
  cout.unsetf(ios::dec); //终止十进制的格式设置
  cout.setf(ios::hex); //设置以十六进制输出格式
  cout<<a<<endl; //以十六进制输出 a
```

```
    return 0;
}
```

程序运行结果如下:

```

-----1-----
123              (域宽为 10, 默认为右对齐)

-----2-----
123              (按数据实际长度输出)

-----3-----
&&&&&&&123        (域宽为 10, 默认为右对齐, 空白处用'&'填充)

-----4-----
123              (按数据实际长度输出, 左对齐)

-----5-----
123.5            (以一般十进制小数形式输出时, 有效数字为 4)

-----6-----
123.4568         (以 fixed 形式输出时, 小数占 4 位)

-----7-----
1.2346e+002&&&&  (用指数格式输出, 域宽为 10, 小数占 4 位, 用'&'填充)

-----8-----
0x15             (十六进制形式, 以 0x 开头)
```

分析以上程序和运行结果。

(1) 在默认情况下, 域宽取值为 0, 这个 0 意味着一个特殊的意义——无域宽, 即数据按自身的宽度打印。

(2) 当用 width 函数设置了域宽后, 只对紧跟着它的第一个输出有影响, 当第一个输出完成后, 域宽又恢复为默认域宽 0。而调用 precision 函数和 fill 函数的设置, 在程序中一直有效, 除非它们被重新设置。setf 函数设置格式后, 如果想改变设置为同组的另一状态, 应当调用 unsetf 函数, 先终止原来的设置状态, 然后再设置其他状态。

(3) 当显示数据所需的宽度比使用 ios::width() 设置的宽度小时, 空余的位置用填充字符来填充, 默认情况下的填充字符是空格。填充字符的填充位置由 ios::left 和 ios::right 规定。若设置 ios::left, 则字符填充在数据右边 (输出数据左对齐); 若设置 ios::right (默认设置), 则字符填充在数据左边 (输出数据右对齐)。

9.2.2 使用预定义的操纵符进行输入输出格式控制

使用 ios 类中的成员函数进行输入输出格式控制时, 每个函数的调用需要写一条语句, 而且不能将它们直接嵌入到输入输出语句中去, 显然使用起来不太方便。C++ 提供了另一种进行输入输出格式控制的方法, 这一方法使用了一种称为操纵符 (也称为操作符或控制符) 的特殊函数。在很多情况下, 使用操纵符进行格式化控制比用 ios 状态标志和成员函数要方便。

操纵符有不带参数的操纵符和带参数的操纵符两类。所有不带形参的操纵符都定义在头文件 iostream.h 中, 而带形参的操纵符则定义在头文件 iomanip.h 中, 因而使用相应的操纵符就必须包含相应的头文件。许多操纵符的功能类似于上面介绍的 ios 类成员函数的功能, 表 9.4 列出了 C++ 提供的预定义操纵符。

表 9.4

C++预定义的操纵符

操纵符	功能	输入/输出
dec	设置整数的基数为 10	用于输入/输出
hex	设置整数的基数为 16	用于输入/输出
oct	设置整数的基数为 8	用于输入/输出
ws	用于在输入时跳过开头的空白符	用于输入
endl	输出一个换行符并刷新输出流	用于输出
ends	插入一个空字符 null, 通常用来结束一个字符串	用于输出
flush	刷新一个输出流	用于输出
setbase(n)	设置整数的基数为 n(n 的取值为 0, 8, 10 或 16), n 的默认值为 0, 即以十进制形式输出	用于输入/输出
setfill(c)	设置 c 为填充字符, 默认时为空格	用于输出
setprecision(n)	设置实数的精度为 n 位, 在以一般十进制小数形式输出时 n 代表有效数字。在以 fixed(固定小数位数)形式和 scientific(指数)形式输出时 n 为小数位数设置域宽为 n	用于输出
setw(n)	设置域宽为 n	用于输出
setiosflags(f)	设置由参数 f 指定的状态标志	用于输入/输出
resetiosflags(f)	终止由参数 f 指定的状态标志	用于输入/输出

操纵符 `setiosflags` 和 `resetiosflags` 要带上状态标志才能使用, 表 9.5 列出了带有常用的状态标志的操纵符 `setiosflags` 和 `resetiosflags`。

表 9.5

部分带有状态标志的操纵符 `setiosflags` 和 `resetiosflags`

操纵符	功能
<code>setiosflags(ios:: left)</code>	数据按域宽左对齐输出
<code>setiosflags(ios:: right)</code>	数据按域宽右对齐输出
<code>setiosflags(ios:: fixed)</code>	固定的小数位数显示
<code>setiosflags(ios:: scientific)</code>	设置浮点数以科学表示法(即指数形式)显示
<code>setiosflags(ios:: showpos)</code>	在正数前添加一个“+”号输出
<code>setiosflags(ios:: uppercase)</code>	在以科学表示法格式 E 和以十六进制输出字母时用大写表示
<code>resetiosflags(f)</code>	终止已设置的状态标志, 在括号中应指定 f 的内容

在进行输入输出时, 操纵符被嵌入到输入或输出链中, 用来控制输入输出的格式, 而不是执行输入或输出操作。为了使用带参数的操纵符, 程序中必须含有下列预编译命令:

```
#include <iomanip>
```

下面通过一些例子来介绍操纵符的使用。

例 9.6 用科学表示法以左对齐方式输出浮点数的值, 并在正数前加上“+”号。

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ double x=45.3;
  cout<<setiosflags(ios::scientific|ios::left|ios::showpos);
  cout<<x<<endl;
  return 0;
```


}

程序运行结果如下：

```
+4.530000e+001
```

例 9.7 在数据符号和数据之间插入指定的填充符。

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ double i=-5.1;
  cout<<setw(10);
  cout<<setfill('*');
  cout<< internal<<i<<endl;
  return 0;
}
```

程序运行结果如下：

```
-*****5.1
```

例 9.8 预定义的操纵符的使用方法举例。

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ cout<<setw(10)<<123<<567<<endl; //①
  cout<<123<<setiosflags(ios::scientific)<<setw(20)<<123.456789<<endl; //②
  cout<<123<<setw(10)<<hex<<123<<endl; //③
  cout<<123<<setw(10)<<oct<<123<<endl; //④
  cout<<123<<setw(10)<<dec<<123<<endl; //⑤
  cout<<resetiosflags(ios::scientific)<<setprecision(4)
    <<123.456789<<endl; //⑥
  cout<<setiosflags(ios::left)<<setfill('#')<<setw(8)<<123<<endl; //⑦
  cout<<resetiosflags(ios::left)<<setfill('$')<<setw(8) <<456<<endl; //⑧
  return 0;
}
```

程序运行结果如下：

```
123567 ①
123 1.234568e+002 ②
123 7b ③
7b 173 ④
173 123 ⑤
123.5 ⑥
123#### ⑦
$$$$$456 ⑧
```

下面分析每条语句和输出结果。

第1条 cout 语句首先设置域宽为 10，之后输出 123 和 567，123 和 567 被连到了一起，所以得到结果①。表明操纵符 setw 只对最靠近它的输出起作用，也就是说，它的作用是“一

次性”的。

第 2 条 cout 语句首先按默认方式输出 123，之后按照浮点数的科学表示法及域宽为 20 输出 123.456789，由于默认时小数位数为 6，所以得到结果②。

第 3 条 cout 语句首先按默认方式输出 123，之后按照域宽为 10，以十六进制输出 123，得到结果③。

第 4 条 cout 语句由于上一条语句中使用了操纵符 hex，其作用仍然保持，所以先输出 123 的十六进制数，之后按照域宽为 10，重新设置进制为八进制，输出 123 得到结果④。结果表明：使用 dec、oct、hex 等操作符后，其作用一直保持，直到重新设置为止。

第 5 条 cout 语句由于上一条语句的操纵符 oct 的作用仍然保持，所以先输出 123 的八进制数，之后按照域宽为 10，用操纵符 dec 恢复进制为十进制后，输出结果⑤。

第 6 条 cout 语句取消浮点数的科学表示法输出后，设置有效数字为 4 位，输出 123.5，从而得到结果⑥。结果表明用 setprecision 操纵符设置有效数字位数后，输出时作四舍五入处理。

第 7 条 cout 语句按域宽为 8，填充字符为“#”，按左对齐输出 123，得到结果⑦。

第 8 条 cout 语句按域宽为 8，填充字符为“\$”，取消左对齐输出(默认对齐方式为右对齐)后，输出 456，得到结果⑧。

9.2.3 使用用户自定义的操纵符进行输入输出格式控制

C++除了提供系统预定义的操纵符外，也允许用户自定义操纵符，合并程序中频繁使用的输入输出操作，使输入输出密集的程序变得更加清晰高效，并可避免意外的错误。下面介绍建立自定义操纵符的方法。

若为输出流定义操纵符函数，则定义形式如下：

```
ostream &操纵符名(ostream &stream)
```

```
{
    自定义代码
    return stream;
}
```

若为输入流定义操纵符函数，则定义形式如下：

```
istream &操纵符名(istream &stream)
```

```
{
    自定义代码
    return stream;
}
```

以上操纵符函数中返回流对象 stream（也可用其他标识符，但与形参表中的流对象必须相同）是一个关键，否则操纵符就不能用在流的输入输出操作序列中。请看以下的例子。

例 9.9 用户自定义操纵符的使用方法举例 1。

```
#include<iostream>
#include<iomanip>
using namespace std;
ostream &outputl(ostream &stream)
{ stream.setf(ios::left);
```


才能向它输出数据。

从操作系统的角度来说,每一个与主机相联的输入输出设备都可以看作是一个文件。例如,键盘是输入文件,显示器和打印机是输出文件。还有磁盘文件、光盘文件和U盘文件等外存文件,其中磁盘文件是使用最广泛的外存文件。由于在程序中对光盘文件和U盘文件的使用方法与磁盘文件相同,为了叙述方便,在本章中凡用到外存文件的地方均以磁盘文件来代表。

操作系统命令一般是将文件作为一个整体来处理的,例如删除文件、复制文件等。由于文件的内容可能千变万化,文件的大小各不相同,为了以统一的方式处理文件,在C++中引入了文件流的概念,文件流是以外存文件为输入输出对象的数据流。输出文件流是从内存流向外存文件的数据,输入文件流是从外存文件流向内存的数据。为了叙述方便,本章中凡用到外存文件的地方均以磁盘文件来代表。

根据文件中数据的组织形式,文件可分为两类:文本文件和二进制文件。文本文件又称ASCII文件,它的每个字节存放一个ASCII代码,代表一个字符。二进制文件则是把内存中的数据,按其在内存中的存储形式原样写到磁盘上存放。假定有一个整数10000,在内存中占两个字节,如果按文本形式输出到磁盘上,则需占5个字节,而如果按二进制形式输出,则在磁盘上只占两个字节。用文本形式输出时,一个字节对应一个字符,因而便于对字符进行逐个处理,也便于输出字符,缺点是占存储空间较多。用二进制形式输出数据,可以节省存储空间和转换时间,但一个字节不能对应一个字符,不能直接以字符形式输出。对于需要暂时保存在外存上,以后又需要输入到内存的中间结果数据,通常用二进制形式保存。

在C++中进行文件操作的一般步骤如下。

- (1) 为要进行操作的文件定义一个流对象。
- (2) 建立(或打开)文件。如果文件不存在,则建立该文件。如果磁盘上已存在该文件,则打开它。
- (3) 进行读写操作。在建立(或打开)的文件基础上执行所要求的输入或输出操作。
- (4) 关闭文件。当完成输入输出操作时,应把已打开的文件关闭。

9.3.2 文件的打开与关闭

1. 文件的打开

在C++中,打开一个文件,就是将这个文件与一个流对象建立关联;关闭一个文件,就是取消这种关联。

为了执行文件的输入输出,C++提供了3个文件流类,如表9.6所示。

表 9.6 用于文件输入输出的文件流类

类 名	说 明	功 能
ifstream	输入文件流类	用于文件的输入
ofstream	输出文件流类	用于文件的输出
fstream	输入输出文件流类	用于文件输入输出

这3个文件流类都定义在头文件 `fstream` 中。

要执行文件的输入输出,须完成以下几件工作。

- (1) 在程序中包含头文件 `fstream`。由于文件的输入输出要用到以上的3个文件流类,而这三个文件流类都定义在 `fstream` 头文件中,所以首先在程序中要包含头文件 `fstream`。

(2) 建立流对象。要以磁盘文件为对象进行输入输出, 必须建立一个文件流类的对象, 通过文件流对象将数据从内存输出到磁盘文件, 或者通过文件流对象从磁盘文件将数据输入到内存。建立流对象的过程就是定义流类的对象, 例如:

```
ifstream in;
ofstream out;
fstream both;
```

分别定义了输入流对象 in, 输出流对象 out, 输入输出流对象 both。其实在用标准设备为对象的输入输出中, 也是要定义对象的, 如 cin 和 cout 就是流对象, C++就是通过流对象进行输入输出的, 由于 cin、cout 已在头文件 iostream 中事先定义, 所以用户不需要自己定义。

(3) 使用成员函数 open 打开文件, 也就是使某一指定的磁盘文件与某一已定义的文件流对象建立关联。

(4) 进行读写操作。在建立(或打开)的文件基础上执行所要求的输入或输出操作。

(5) 使用 close 函数将打开的文件关闭。

open 函数是上述 3 个流类的成员函数, 其原型是在 fstream 中定义的。在 ifstream、ofstream 和 fstream 类中均有定义。调用成员函数 open 的一般形式为:

文件流对象.open(文件名, 使用方式);

其中“文件名”可以包括路径(如“d:\c++\file1.dat”), 如缺省路径, 则默认为当前目录下的文件。“使用方式”决定文件将如何被打开, 如表 9.7 所示。

表 9.7 文件的使用方式

方 式	功 能
ios::in	以输入方式打开文件
ios::out	以输出方式打开文件, 如果已有此名字的文件, 则将其原有的内容全部清除
ios::app	以输入方式打开文件, 写入的数据添加到文件尾部
ios::ate	打开一个已有的文件, 把文件指针移到文件末尾
ios::trunc	打开一个文件, 若文件已存在, 删除其中全部数据, 若文件不存在, 则建立新文件。如已指定了 ios::out 方式, 而未指定 ios::app、ios::in, 则同时默认此方式
ios::nocreate	打开一个已有的文件, 若文件不存在, 则打开失败
ios::noreplace	打开一个文件, 若文件不存在, 则建立新文件; 若文件存在, 则打开失败
ios::binary	以二进制方式打开一个文件, 如不指定此方式, 则默认为文本文件

下面对打开方式作进一步的说明。

(1) 新版本的 C++ 系统 I/O 类库中不提供 ios::nocreate 和 ios::noreplace。

(2) 每一个打开的文件都有一个文件指针, 该指针的初始位置由输入/输出的方式指定, 每次读写都从文件指针的当前位置开始。每读一个字节, 指针就后移一个字节。当文件指针移到最后, 就会遇到文件结束符 EOF (文件结束符也占一个字节, 其值为-1), 此时流对象的成员函数 eof 的值为非 0 值(一般设为 1), 表示文件结束了。

(3) 如果希望向文件尾部添加数据, 则应当用“ios::app”方式打开文件, 但此时文件必须存在。打开时, 文件位置指针移到文件尾部。用这种方式打开的文件只能用于输出。

(4) 用“ios::ate”方式打开一个已存在的文件时, 文件位置指针自动移到文件的尾部, 数据可以写入文件中的任何地方。

(5) 用“ios::in”方式打开的文件只能用于输入数据, 而且该文件必须已经存在。

(6) 通常, 当用 `open` 函数打开文件时, 如果文件存在, 则打开该文件, 否则建立该文件。但当用 “`ios::nocreate`” 方式打开文件时, 表示不建立新文件, 在这种情况下, 如果要打开的文件不存在, 则函数 `open` 调用失败。相反, 如果使用 “`ios::noreplace`” 方式打开文件, 则表示不修改原来文件, 而是要建立新文件。因此, 如果文件已经存在, 则 `open` 函数调用失败。

(7) 当使用 “`ios::trunk`” 方式打开文件时, 如果文件已存在, 则清除该文件的内容, 文件长度被压缩为零。实际上, 如果指定 “`ios::out`” 方式, 且未指定 “`ios::ate`” 方式或 “`ios::app`” 方式, 则隐含为 “`ios::trunk`” 方式。

(8) 如果使用 “`ios::binary`” 方式, 则以二进制方式打开文件, 默认时, 所有的文件以文本方式打开。

了解了文件的使用方式后, 可以通过以下步骤打开文件:

(1) 定义一个流类的对象, 例如:

```
ofstream out;
```

定义了类 `ofstream` 的对象 `out`, 它是一个输出流对象。

(2) 使用 `open` 函数打开文件, 也就是使某一文件与上面定义的流对象建立关联。例如:

```
out.open("test.dat", ios::out);
```

表示调用成员函数 `open`, 使文件流对象 `out` 与文件 `test.dat` 建立关联, 即打开磁盘文件 `test.dat`, 并指定它为输出文件, 文件流对象 `out` 将向磁盘文件 `test.dat` 输出数据。`ios::out` 表示以输出方式打开一个文件。或者简单地说, 此时 `test.dat` 是一个输出文件, 接收从内存输出的数据。

以上是打开文件的一般操作步骤。实际上, 由于文件的输入输出方式参数有默认值, 对于类 `ifstream`, 文件使用方式的默认值为 `ios::in`; 而对于类 `ofstream`, 文件使用方式的默认值为 `ios::out`。

因此, 上述语句通常可写成:

```
out.open("test.dat");
```

当一个文件需要用两种或多种方式打开时, 可以用 “位或” 操作符 (即 “`|`”) 把几种方式组合在一起。例如, 为了打开一个能用于输入和输出的二进制文件, 则可以采用以下方法打开文件:

```
fstream mystream;
mystream.open("test.dat", ios::in|ios::out|ios::binary);
```

还可以举出一些用 “位或” 操作符把几种方式组合在一起的例子:

```
ios::in|ios::out           //以输入和输出方式打开文件, 文件可读可写
ios::out|ios::binary       //以二进制方式打开一个输出文件
ios::in|ios::binary       //以二进制方式打开一个输入文件
ios::in|ios::nocreate      //打开一个输入文件, 若文件不存在
                           //则返回打开失败的信息
ios::app|ios::nocreate     //打开一个输出文件, 在文件尾接着写数据
                           //若文件不存在, 则返回打开失败的信息
```

在实际编程时, 还有一种打开文件的方法, 即在定义文件流对象时指定参数, 通过调用

文件流类的构造函数来实现打开文件的功能，例如：

```
ofstream out("test.dat");
```

因为定义文件流类 `ifstream`、`ofstream` 与 `fstream` 的对象时，都能自动打开文件流类的构造函数，这些构造函数的参数及默认值与 `open` 函数的完全相同。这是打开一个文件的最常见的形式，使用起来比较方便。以上打开文件的语句相当于：

```
ofstream out;
out.open("test.dat");
```

如果文件打开操作失败，`open` 函数的返回值为 0（假），如果是用调用构造函数的方式打开文件的，则与文件相联系的流对象的值为 0。因此，无论是使用构造函数来打开文件，还是直接调用函数 `open` 来打开文件，通常都要测试打开文件是否成功。可以使用类似下面的方法进行检测：

```
if (!out)
{ cout<<"Cannot open file! \n";
  //错误处理代码
}
```

2. 文件的关闭

输入输出操作完成后，应该将文件关闭。关闭文件可使用 `close` 函数完成，`close` 函数也是流类中的成员函数，它不带参数，例如：

```
out.close();
```

就将与流对象 `out` 所关联的磁盘文件关闭了。

关闭实际上就是将所打开的磁盘文件与流对象“脱钩”，这样，就不能通过文件流对象对该文件进行输入或输出操作了。此时可以将文件流对象与其他磁盘文件建立关联，通过文件流对象对新的文件进行输入或输出。例如：

```
out.open("test2.dat", ios::out);
```

此时文件流对象又与 `test2.dat` 建立了关联，即打开磁盘文件 `test2.dat`，并指定它为输出文件。

在进行文件操作时，应养成将已完成操作的文件关闭的习惯。如果不关闭文件，则有可能丢失数据。

9.3.3 文本文件的读写

一旦文件打开了，从文件中读取文本数据与向文件中写入文本数据都十分容易。流类库的输入输出操作 `<<`、`put`、`write`、`>>`、`get`、`getline` 等，同样可以用于文本文件的输入输出。

例 9.11 把字符串 “I am a student.”，写入磁盘文件 `test1.dat` 中。

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ ofstream fout1("test1.dat", ios::out);

    //定义输出文件流对象 fout1，打开输入文件 test1.dat

    if (!fout1)
        //如果文件打开失败，fout1 返回 0 值
```

```

{ cout<<"Cannot open output file.\n";
  exit(1);
}
foutl<<"I am a student."; //把一个字符串写到磁盘文件 test1.dat 中

foutl.close(); //将与流对象 foutl 所关联的输入文件 test1.dat 关闭
return 0;
}

```

程序运行后，屏幕上不显示任何信息，因为输出的内容存入文件 test1.dat 中。可以利用 Windows 的 Word 或 Dos 的 TYPE 命令打开文件 test1.dat，该文件的内容如下：

```
I am a student.
```

说明：

语句“ofstream foutl("test1.dat", ios::out);”中的参数 ios::out 可以省略，如不写此项，则默认为 ios::out。以下两种写法是等价的：

```

ofstream foutl("test1.dat", ios::out);
ofstream foutl("test1.dat");

```

例 9.12 把磁盘文件 test1.dat 中的内容读出并显示在屏幕上。

```

include<iostream>
#include<fstream>
using namespace std;
int main()
{ ifstream finl("test1.dat", ios::in); //定义输入文件流对象 finl
  //打开输入文件 test1.dat
  if (!finl) //如果文件打开失败，finl 返回 0 值
  { cout<<"Cannot open output file.\n";
    exit(1);
  }
  char str[80];
  finl.getline(str, 80); //从磁盘文件 test1.dat 读入字符串
  //赋给字符数组 str
  cout<<str<<endl; //屏幕上显示出 str 的值
  finl.close(); //将与流对象 finl 所关联的输入文件 test1.dat 关闭
  return 0;
}

```

说明：

语句“ifstream finl("test1.dat", ios::in);”中的参数 ios::in 可以省略，如不写此项，则默认为 ios::in。以下两种写法是等价的：

```

ifstream finl("test1.dat", ios::in);
ifstream finl("test1.dat");

```

例 9.13 把一个整数、一个浮点数和一个字符串写到磁盘文件 fl.dat 中。

```

#include<iostream>
#include<fstream>
using namespace std;
int main()

```



```

{ ofstream fout("f1.dat", ios::out);
    //定义输出文件流对象 fout, 打开输出文件 f1.dat
    if (!fout)
        //如果文件打开失败, fout 返回 0 值
    { cout<<"Cannot open output file.\n, ";
        exit(1);
    }
    fout<<10<<" "<<123.456<<"\nThis is a text file.\n\n";
    //把一个整数、一个浮点数和一个字符串写到磁盘文件 f1.dat 中
    fout.close();
    //将与流对象 fout 所关联的输出文件 f1.dat 关闭
    return 0;
}

```

程序运行后, 屏幕上不显示任何信息, 因为输出的内容存入文件 f1.dat 中。可以利用 Windows 的 Word 或 Dos 的 TYPE 命令打开文件 f1.dat, 该文件的内容如下:

```
10 123.456 "This is a test file"
```

下面再看一个对同一个文件进行输出和输入的文件。

例 9.14 先建立一个输出文件, 向它写入数据, 然后关闭文件, 再按输入模式打开它, 并读取信息。

```

#include<iostream>
#include<fstream>
using namespace std;
int main()
{ ofstream fout("f2.dat", ios::out);
    //定义输出文件流对象 fout, 打开输出文件 f2.dat
    if (!fout)
        //如果文件打开失败, fout 返回 0 值
    { cout<<"Cannot open output file.\n";
        return 1;
    }
    fout<<100<<" " <<hex<<100<<endl;
    //把一个十进制整数和一个十六进制
    //整数写到磁盘文件 f2.dat 中
    fout<<"\nHello!\n\n";
    //把一个字符串写到磁盘文件 f2.dat 中
    fout.close();
    //将与流对象 fout 所关联的输出文件 f2.dat 关闭
    ifstream fin("f2.dat", ios::in);
    //定义文件流对象 fin, 打开输入文件 f2.dat
    if (!fin)
        //如果文件打开失败, fin 返回 0 值
    { cout<<"Cannot open input file.\n";
        return 1;
    }
    char str[80];
    while (fin)
    {
        fin.getline(str, 80);
        //从磁盘文件 f2.dat 读入信息
        //赋给字符数组 str
        cout<<str<<endl;
    }
    fin.close();
    //将与流对象 fin 所关联的输入文件 f2.dat 关闭
    return 0;
}

```

程序运行后, 首先建立一个输出文件 `f2.dat`, 并向它写入数据。完成写入数据后, 关闭输出文件 `f2.dat` 后, 再将文件 `f2.dat` 按输入模式打开, 并从磁盘文件 `f2.dat` 读入信息赋给字符数组 `str`。最后在屏幕上显示出 `str` 的值, 如下所示:

```
100 64
"Hello!"
```

可以看到, 在这个例子中, 首先定义输出文件流对象 `fout`, 并使它与文件 `f2.dat` 建立关联, 即打开磁盘文件 `f2.dat`, 并指定它为输出文件, 完成输出操作后, 关闭文件 `f2.dat`。接着, 定义输入文件流对象 `fin`, 并使它与文件 `f2.dat` 建立关联, 即打开磁盘文件 `f2.dat`, 并指定它为输入文件, 完成输入操作后, 再关闭文件 `f2.dat`。

9.3.4 二进制文件的读写

前面已经介绍, 文件可分为文本文件和二进制文件。文本文件又称 ASCII 文件, 它的每个字节存放一个 ASCII 代码, 代表一个字符。二进制文件则是把内存中的数据, 按其在内存中的存储形式原样写到磁盘上存放。

最初设计流的目的是用于文本, 因此在默认情况下, 文件用文本方式打开。在以文本模式输出时, 若遇到换行符 (十进制 10) 便自动被扩充为回车换行符 (十进制 13 和 10)。如果所操作的文件不是普通的文本文件, 文件中包含了一些控制符, 比如换行符或文件结束符, 这种自动扩充有时可能使文件处理发生问题, 请看以下程序:

```
#include<fstream>
using namespace std;
int iarray[2]={65, 10};
int main()
{ ofstream fout("f3.dat", ios::out);
  fout.write((char*) iarray, sizeof(iarray));
  fout.close();
  return 0;
}
```

当执行程序, 向文件中输出时, ASCII 值 10 会被自动转换成 ASCII 值 13 (CR) 及 10 (LF)。然而这里的转换显然不是我们所需要的。要想解决这一问题, 就要采用二进制模式输出。使用二进制模式输出时, 其中所写的字符是不转换的。

对于二进制文件的操作也需要先打开文件, 操作结束后要关闭文件。在打开文件时要用 “`ios::binary`” 指定为以二进制形式传送和存储。

对二进制文件进行读写有两种方式, 其中一种使用的是函数 `get` 和 `put`, 另一种使用的是函数 `read` 和 `write`。这四种函数也可以用于文本文件的读写。在此主要介绍对二进制文件的读写。除字符转换方面略有差别外, 文本文件的处理过程与二进制文件的处理过程基本相同。

1. 用 `get` 函数和 `put` 函数读写二进制文件

前面我们已经介绍过, `get` 函数是输入流类 `istream` 中定义的成员函数, 它可以与从流对象连接的文件中读出数据, 每次读出一个字节 (字符)。 `put` 函数是输出流类 `ostream` 中的成员函数, 它可以向与流对象连接的文件中写入数据, 每次写入一个字节 (字符)。

下面再举一个使用 `get` 和 `put` 函数读写二进制文件的例子。

例 9.15 将'a'至'z'的26个英文字母写入文件，而后从该文件中读出并显示出来。

```
#include<iostream>
#include<fstream>
using namespace std;
int test_put()
{ ofstream outf("f3.dat", ios::binary);
    //定义输出文件流对象 outf, 打开二进制输出文件 f3.dat

    if (!outf)
        //如果文件打开失败, outf 返回 0 值
    { cout<<"Cannot open output file\n, ";
      exit(1);
    }
    char ch='a';
    for (int i=0;i<26;i++)
    { outf.put(ch);
      ch++;
    }
    outf.close();
    return 0;
}

int test_get()
{ ifstream inf("f3.dat", ios::binary);
    //定义输入文件流对象 inf, 打开二进制输入文件 f3.dat

    if (!inf)
        //如果文件打开失败, inf 返回 0 值
    { cout<<"Cannot open input file\n, ";
      exit(1);
    }
    char ch;
    while(inf.get(ch))
        cout<<ch;
    inf.close();
    return 0;
}

int main()
{ test_put();
  test_get();
  return 0;
}
```

程序运行结果如下：

```
abcdefghijklmnopqrstuvwxyz
```

该程序中，先调用函数 test_put，以输出方式打开二进制文件 f3.dat，然后通过 put 函数将'a'至'z'的26个英文字母写入文件 f3.dat，中，再关闭文件。接着调用函数 test_get，再次以输入方式打开二进制文件 f3.dat，然后通过 get 函数把文件 f3.dat 中的26个字符读到 ch 中，并在屏幕上显示出来。

2. 用 read 函数和 write 函数读写二进制文件

有时需要读写一组数据（如一个结构变量的值），为此 C++ 提供了两个函数 read 和 write，用来读写一个数据块，这两个函数最常用的调用格式如下：

```
inf.read(char *buf, int len)
outf.write(const char *buf, int len)
```

`read` 是流类 `istream` 中的成员函数，其有两个参数：第 1 个参数 `buf` 是一个指针，它指向读入数据所存放的内存空间的起始地址；第 2 个参数 `len` 是一个整数值，它是要读入的数据的字节数。`read` 函数的功能是从与输入文件流对象 `inf` 相关联的磁盘文件中，读取 `len` 个字节（或遇 EOF 结束），并把它们存放在字符指针 `buf` 所指的一段内存空间内。如果在 `len` 个字节（字符）被读出之前就达到了文件尾，则 `read` 函数停止执行。

`write` 是流类 `ostream` 的成员函数，参数的含义及调用注意事项与 `read` 函数类似。`write` 函数的功能是将字符指针 `buf` 所给出的地址开始的 `len` 个字节的内容不加转换地写到与输出文件流对象 `outf` 相关联的磁盘文件中。

注意，第 1 个参数的数据类型为 `char*`，如果是其他类型的数据，必须进行类型转换，例如：

```
int array[]={50, 60, 70};
read((char*)& array, sizeof (array));
```

此例定义了一个整型数组 `array`，为了读入它的全部数据，必须在 `read` 函数中给出它的首地址，并把它转换为 `char*` 类型。由 `sizeof` 函数确定要读入的字节数。

例 9.16 将两门课程的课程名和成绩以二进制形式存放在磁盘文件中。

```
#include<iostream>
#include<fstream>
using namespace std;
struct list
{ char course[15];
  int score;
};
int main()
{ list list1[2]={"Computer", 90, "Mathematics", 78};
  ofstream out("f4.dat", ios::binary);
                                     //定义输出文件流对象 out，打开二进制输出文件 f4.dat
  if (!out)                          //如果文件打开失败，out 返回 0 值
  { cout<<"Cannot open output file.\n";
    abort();                          //退出程序，其作用与 exit 相同
  }
  for (int i=0;i<2;i++)
    out.write((char*) &list1[i], sizeof(list1[i]));
  out.close();
  return 0;
}
```

程序执行后，屏幕上不显示任何信息，但程序已将两门课程的课程名和成绩以二进制形式写入文件 `f4.dat` 中。用下面的程序可以读取文件 `f4.dat` 中的数据，并在屏幕上显示出来，以验证前面程序的操作。

例 9.17 将例 9.16 以二进制形式存放在磁盘文件中的数据（两门课程的课程名和成绩）读入内存，并在显示器上显示。

```
#include<iostream>
#include<fstream>
using namespace std;
struct list
```

```

{ char course[15];
  int score;
};
int main()
{ list list2[2];
  ifstream in("f4.dat", ios::binary);
                                     //定义输入文件流对象 in, 打开二进制输入文件 f4.dat
  if (!in)                           //如果文件打开失败, in 返回 0 值
  { cout<<"Cannot open input file.\n";
    abort();                          //退出程序, 其作用与 exit 相同
  }
  for (int i=0; i<2; i++)
  { in.read((char *) &list2[i], sizeof(list2[i]));
    cout<<list2[i].course<<" "<<list2[i].score<<endl;
  }
  in.close();
  return 0;
}

```

程序运行后在显示器上显示:

Computer 90

Mathematics 78

3. 检测文件结束

在文件结束的地方有一个标志位, 记为 EOF(end of file)。采用文件流方式读取文件时, 使用成员函数 eof(), 可以检测到这个结束符。如果该函数的返回值非零, 表示到达文件尾。返回值为零表示未到达文件尾。该函数的原型是:

```
int eof();
```

函数 eof() 的用法示例如下:

```
ifstream ifs;
```

```
if (!ifs.eof())           //尚未到达文件尾
```

还有一个检测方法就是检查该流对象是否为零, 为零表示文件结束:

```
ifstream ifs;
```

```
if (!ifs)                 //尚未到达文件尾
```

也许读者注意到我们在例 9.1 中使用了以下检测流对象到达末尾的方法, 下面将这个例子简要地重述一下:

```
while(cin.get(ch))
cout.put(ch);
```

这是一个很通用的方法, 就是检测文件流对象的某些成员函数的返回值是否为 0, 为 0 表示该流 (亦即对应的文件) 到达了末尾。

当从键盘上输入字符时, 其结束符是 ctrl_z, 也就是说, 按下 ctrl_z, eof() 函数返回的值为真。

4. 二进制数据文件的随机读写

前面介绍的文件操作都是按一定顺序进行读写的，因此称为顺序文件。对于顺序文件而言，只能按实际排列的顺序，一个一个地访问文件中的各个元素。为了增加对文件访问的灵活性，C++系统总是用读或写文件指针记录着文件的当前位置，在类 `istream` 及类 `ostream` 中定义了几个与读或写文件指针相关的成员函数，使我们可以在输入输出流内随机移动文件指针，从而可以对文件的数据进行随机读写。

类 `istream` 提供了 3 个成员函数来对读指针进行操作，它们是：

<code>tellg()</code>	返回输入文件读指针的当前位置
<code>seekg(文件中的位置)</code>	将输入文件中读指针移到指定的位置
<code>seekg(位移量, 参照位置)</code>	以参照位置为基准移动若干字节

函数参数中的“文件中的位置”和“位移量”都是 `long` 型整数，以字节为单位。“参照位置”可以是下面的三者之一：

<code>ios::beg</code>	从文件开头计算要移动的字节数
<code>ios::cur</code>	从文件指针的当前位置计算要移动的字节数
<code>ios::end</code>	从文件末尾计算要移动的字节数

例如，假设 `inf` 是类 `istream` 的一个流对象，则

```
inf.seekg(-50, ios::cur);
```

表示使输入文件中的读指针以当前位置为基准向前(文件的开头方向)移动 50 字节。

```
inf.seekg(50, ios::beg);
```

表示使输入文件中的读指针从文件的开头位置后移 50 字节。

```
inf.seekg(-50, ios::end);
```

表示使输入文件中的读指针从文件的末尾位置前移 50 字节。

类 `ostream` 提供了 3 个成员函数来对写指针进行操作，它们是：

<code>tellp()</code>	返回输出文件写指针的当前位置
<code>seekp(文件中的位置)</code>	将输出文件中写指针移到指定的位置
<code>seekp(位移量, 参照位置)</code>	以参照位置为基准移动若干字节

这 3 个成员函数的含义与前面讲过的操作读指针的 3 个成员函数的含义相似，只是它们用来操作写指针。

函数 `seekg` 和 `seekp` 的第 2 个参数可以省略，在这种情况下，就是默认 `ios::beg`，即从文件的开头来计算要移动的字节数。例如：

```
inf.seekg(50);
```

表示使输入文件中的读指针从文件的开头位置后移 50 个字节。

注意，以上几个函数的命名是有一定的规律的。由于“g”是 `get` 的第 1 个字母，因此带 `g` 的函数（如函数 `tellg` 和 `seekg`）是用于输入的函数。而由于“p”是 `put` 的第 1 个字母，因此带 `p` 的函数（如函数 `tellp` 和 `seekp`）是用于输出的函数。

例 9.18 随机访问二进制数据文件。

有3门课程的数据,要求:

- (1) 以读写方式打开一个磁盘文件,并把这些数据存到磁盘文件中;
- (2) 将文件指针定位到第3门课程,读取第3门课程的数据并显示出来;
- (3) 将文件指针定位到第1门课程,读取第1门课程的数据并显示出来;
- (4) 将文件指针从当前位置定位到下一门课程,读取该门课程的数据并显示出来。

```
#include<iostream>
#include<fstream>
using namespace std;
struct List
{ char course[15];
  int score;
};
int main()
{ List list3[3]={{"Computer", 90}, {"Mathematics", 78}, {"English", 84}};
  List st;
  fstream ff("f6.dat", ios::out|ios::in|ios::binary);
                                     //定义类fstream的流对象ff,以读写方式打开二进制文件f6.dat

  if(!ff)
  { cout<<"open f6.dat error!"<<endl;
    abort();                          //退出程序
  }

  for (int i=0;i<3;i++) //将3门课程的数据写入已存在的磁盘文件f6.dat中
  ff.write((char*)&list3[i], sizeof(List));

  ff.seekp(sizeof(List)*2);           //将文件指针定位到第3门课程
  ff.read((char*)&st, sizeof(List));   //读取第3门课程的数据
  cout<<st.course<<"\t"<<st.score<<endl; //显示第3门课程的数据
  ff.seekp(sizeof(List)*0);           //将文件指针定位到第1门课程
  ff.read((char*)&st, sizeof(List));   //读取第1门课程的数据
  cout<<st.course<<"\t"<<st.score<<endl; //显示第1门课程的数据
  ff.seekp(sizeof(List)*1, ios::cur); //将文件指针从当前位置定位到下一门课程
  ff.read((char*)&st, sizeof(List));   //读取该门课程的数据
  cout<<st.course<<"\t"<<st.score<<endl; //显示该门课程的数据
  ff.close();                          //关闭文件
  return 0;
}
```

程序运行结果如下:

English 84	(显示第3门课程的数据)
Computer 90	(显示第1门课程的数据)
English 84	(显示第2门课程的数据)

9.4 应用举例

例 9.19 将一个二进制文件中的所有数字(0~9)读出并拷贝到另一个二进制文件中去。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
```

```

class Copy_file{
public:
    Copy_file();                //打开源文件，建立目的文件
    ~Copy_file();               //关闭源、目的文件
    void Copy_files();           //读源文件，将其内的小写字母字符写入到目的文件中
    void in_file();              //声明函数 in_file 的原型，输出源文件内容
    void outfile();              //声明函数 outfile 的原型，输出目的文件内容
private:
    fstream inf;                 //用 fstream 类定义输入输出流对象，用来关联源文件
    fstream outf;                //用 fstream 类定义输入输出流对象，用来关联目的文件
    char file1[20];              //存放源文件名
    char file2[20];              //存放目的文件名
};

Copy_file::Copy_file()
{ cout<<"请输入源文件名:";
  cin>>file1;
  inf.open(file1, ios::in|ios::binary); //打开源文件
  if(!inf)
  { cout<<"不能打开源文件:"<<file1<<endl;
    abort();
  }
  cout<<"请输入目的文件名:";
  cin>>file2;
  outf.open(file2, ios::in|ios::out|ios::binary); //打开目的文件
  if(!outf)
  { cout<<"不能打开目的文件:"<<file2<<endl;
    abort();
  }
}

Copy_file::~Copy_file() //关闭源文件和目的文件
{ inf.close();
  outf.close();
}

void Copy_file::Copy_files() //从源文件中读出字符，并将小写字母字符写入目的文件中
{
    char ch;
    inf.seekg(0);
    inf.get(ch);
    while(!inf.eof())
    {
        if(ch>='0'&&ch<='9')
            outf.put(ch);
        inf.get(ch);
    }
}

void Copy_file::in_file() //定义函数 in_file，输出源文件内容
{ char ch;
  inf.close();
  inf.open(file1, ios::in|ios::binary);
  inf.get(ch);
  while(!inf.eof())

```



```

    { cout<<ch;
      inf.get(ch);
    }
    cout<<endl;
}
void Copy_file::outfile()           //定义函数 outfile, 输出目的文件内容
{ char ch;
  outf.seekg(0);                  //使文件指针定位在文件的首位
  outf.get(ch);
  while(!outf.eof())
  { cout<<ch;
    outf.get(ch);
  }
  cout<<endl;
}
int main()
{ Copy_file cf;
  cf.Copy_files();
  cout<<"源文件中内容:"<<endl;    //输出源文件中的内容
  cf.in_file();
  cout<<"目的文件中内容:"<<endl;  //输出目的文件中的内容
  cf.outfile();
  return 0;
}

```

程序运行结果如下:

请输入源文件名: ss.txt

请输入目的文件名: oo.txt

源文件中内容:

10 123.456"This is a text file."

目的文件中内容:

10123456

说明:

假设本例中源文件 ss.txt 已经存在, 文件中的内容为:

10 123.456"This is a text file."

实 验

实验目的和要求

1. 学习 C++ 格式化的输入输出方法。
2. 学习 C 重载运算符 "<<" 和 ">>" 的方法。
3. 学习 C 磁盘文件的输入输出方法。

实验内容和步骤

1. 下面给出的程序 test9_1_1.cpp 用于打印九九乘法表, 但程序中存在错误。请上机调

试,使得此程序运行后,能够输出如下所示的九九乘法表。

```
* 1 2 3 4 5 6 7 8 9
1 1
2 2 4
3 3 6 9
4 4 8 12 16
5 5 10 15 20 25
6 6 12 18 24 30 36
7 7 14 21 28 35 42 49
8 8 16 24 32 40 48 56 64
9 9 18 27 36 45 54 63 72 81
```

```
//test9_1_1.cpp
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{ int i, j ;
  cout << " ";
  for(i=1; i<=9; i++)
    cout<<i<<" ";
  cout<<endl;
  for(i=1;i<=9; i++)
  { cout<<i;
    for(j=1; j<=i; j++)
      cout<<i*j;
    }
  return 0;
}
```

2. 下面的程序用于统计文件 xyz.txt 中的字符个数,请填空完成程序。

```
//test9_2_1.cpp
#include<iostream>
#include<fstream>
using namespace std;
int main()
{ char ch;
  int i=0;
  ifstream file;
  file.open("xyz.txt", ios::in);
  if( ① )
  {
    cout<<"xyz.txt cannot open"<<endl;
    abort();
  }
  while(!file.eof())
  {
    ②
    i++;
  }
  cout<<"文件字符个数: "<<i<<endl;
  ③
  return 0;
}
```

3. 重载运算符“<<”和“>>”，使其能够输入一件商品的信息和输出这件商品的信息。商品的信息有编号、商品名和价格。假如商品类 `Merchandise` 的框架如下：

```
class Merchandise{
public:
    Merchandise();
    ~Merchandise();
    friend istream& operator>>(istream& in, Merchandise& s);
                                //输入一件商品的信息
    friend ostream& operator<<(ostream& out, Merchandise& s);
                                //输出这件商品的信息
private:
    int no;                    //编号
    char* name;                //商品名
    double price;              //价格
};
```

要求实现该类，并编写以下的 `main` 函数对该类进行操作。

```
int main()
{ Merchandise mer;
  cin>>mer;
  cout<<mer;
  return 0;
}
```

4. 编写一个程序，将两个文本文件连接成一个文件，然后将此文件中所有小写字母转换成大写字母，并打印出来。

习 题

- 【9.1】基类 `ios` 有哪 4 个直接派生类？
- 【9.2】什么叫流对象？C++ 有哪 4 个预定义的流对象？它们分别与什么具体设备相关联？
- 【9.3】`cerr` 和 `clog` 之间的区别是什么？
- 【9.4】C++ 提供了哪两种控制输入输出格式的方法？
- 【9.5】在 C++ 中进行文件操作的一般步骤是什么？
- 【9.6】根据文件中数据的组织形式，文件可分为哪两类？这两类有什么区别？
- 【9.7】有如下程序：

```
#include<iostream>
using namespace std;
int main()
{ int i=100;
  cout.setf(ios::hex);
  cout<<i<<"\t";
  cout<<i<<"\t";
  cout.setf(ios::dec);
  cout<<i<<"\n";
  return 0;
}
```

}

以上程序运行的结果是 ()。

- A. 64 100 64 B. 64 64 64
C. 64 64 100 D. 64 100 100

【9.8】C++提供的预定义操纵符中, () 是转换为十六进制形式的操纵符。

- A. dec B. oct C. hex D. left

【9.9】控制格式 I/O 的操作中, () 是设置域宽的。

- A. setw() B. oct C. setfill() D. ws

【9.10】进行文件操作时需要包含文件 ()。

- A. string.h B. fstream.h C. stdio.h D. stdlib.h

【9.11】磁盘文件操作中的访问模式常量 () 是以追加方式打开文件的。

- A. app B. out C. in D. ate

【9.12】当使用 ifstream 流类定义一个流对象并打开一个磁盘文件时, 文件的隐含打开方式是 ()。

- A. ios::trunk B. ios::out C. ios::in D. ios::binary

【9.13】下列函数中, () 是对文件进行写操作的。

- A. seek() B. read() C. get() D. put()

【9.14】使用 myFile.open("Sales.dat", ios::app); 语句打开文件 Sales.dat 后, 则 ()。

- A. 该文件只能用于输出
B. 该文件只能用于输入
C. 该文件既可以用于输出, 也可以用于输入
D. 若该文件存在, 则清除该文件的内容

【9.15】如果输入是 abcdefgh, 希望输出 abcd efgh, 请填写程序中 width 的宽度。

```
#include<iostream>
#include <iomanip>
using namespace std;
int main()
{ char buffer1[10], buffer2[10];
  cin.width( );
  cin>>buffer1>>buffer2;
  cout.width( );
  cout<<buffer1<<' '<<buffer2<<"\n";
  return 0;
}
```

【9.16】分别计算 5! 到 9! 的值, 使用 setw() 控制 “=” 左边的数值宽度。

【9.17】编写一个程序, 打印 2~10 之间的数字的自然对数与以 10 为底的对数。对表进行格式化, 使得数字可以在域宽为 10 的范围内, 用 5 个十进制位置的精度进行右对齐。

【9.18】写一个程序, 用于统计某文本文件中单词 is 的个数。

【9.19】编写一个程序, 要求定义 in 为 fstream 的对象, 与输入文件 file1.txt 建立关联, 文件 file1.txt 的内容如下:

```
abcdef
ghijklmn
```

定义 out 为 fstream 的对象，与输出文件 file2.txt 建立关联。当文件打开成功后将 file1.txt 文件的内容转换成大写字母，输出到 file2.txt 文件中。

【9.20】编写一个程序，要求定义 in 为 fstream 的对象，与输入文件 file1.txt 建立关联，文件 file1.txt 的内容如下：

```
aabbcc
```

定义 out 为 fstream 的对象，与输出文件 file2.txt 建立关联。当文件打开成功后将 file1.txt 文件的内容附加到 file2.txt 文件的尾部。运行前 file2.txt 文件的内容如下：

```
ABCDEF  
GHIJKLMN
```

运行后，再查看文件 file2.txt 的内容。

第 10 章

异常处理和命名空间

异常处理和命名空间是 C++ 发展后期增加的新功能，以帮助程序设计人员更方便地进行程序的设计和调试工作。异常处理是对所能预料的运行错误进行处理的一套实现机制。有了异常处理，C++ 程序可以在环境出现意外或用户操作不当的情况下，作出正确合适的处理和防范。所谓命名空间，实际上就是一个由程序设计者命名的内存区域。C++ 引入了命名空间，可以用来处理程序中常见的同名冲突问题。

10.1 异常处理

由于程序中存在的缺陷，或者由于程序的执行环境出现例外，或者由于其他未曾预料到的情况，程序在运行过程中经常会出现一些异常。一个程序不仅要在正常的环境下运行正确，而且在环境出现意外或用户操作不当的情况下，也应该有正确合适的处理和防范。C++ 提供了专门的异常处理机制。异常处理是对所能预料的运行错误进行处理的一套实现机制。

10.1.1 异常处理概述

程序中的错误，分为编译时的错误和运行时的错误。编译时的错误主要是语法错误，如关键字拼写错误、语句末尾缺分号、括号不匹配等。这类错误相对比较容易修正，因为编译系统会指出在第几行，是什么样的错误。运行时的错误则不然，其中有些错误甚至是不可预料的，如算法出错；有些虽然可以预料但却无法避免，如内存空间不够，无法实现指定的操作等；还有在函数调用时存在的一些错误，如无法打开输入文件、数组下标越界等。如果在程序中没有对这些错误的防范措施，往往得不到正确的运行结果甚至导致程序不正常终止，或出现死机现象。这类错误比较隐蔽，不易被发现，是程序调试中的一个难点。

程序在运行过程中出现的错误统称为异常，对异常的处理称为异常处理。我们在设计程序时，应当事先分析程序运行时可能出现的各种意外情况，并且分别制定出相应的处理方法，使程序能够继续执行，或者至少给出适当的提示信息。传统的异常处理方法基本上是采取判断或分支语句来实现，如例 10.1 所示。

例 10.1 传统的异常处理方法举例。

```
#include<iostream>
using namespace std;
int Div(int x, int y);    //函数 Div 的原型
int main()
```

```

{ cout<<"7/3="<<Div(7, 3)<<endl;
  cout<<"5/0="<<Div(5, 0)<<endl;
  return 0;
}

int Div(int x, int y)          //定义函数 Div
{ if (y==0)
  { cout<<"除数为 0, 错误!"<<endl;
    exit (0);
  }
  return x/y;
}

```

程序运行结果如下:

```
7/3=2
```

```
除数为 0, 错误!
```

在这个例子中, 函数 Div 用来计算 x/y 的值。当调用函数时, 一旦除数 y 为 0, 则程序输出提示信息“除数为 0, 错误!”, 然后退出程序的运行。

传统的异常处理方法可以满足小型的应用程序需要。但是在一个大型软件系统中, 包含许多模块, 每个模块又包含许多函数, 函数之间又互相调用, 比较复杂。如果在每一个函数中都设置处理异常的程序段, 会使程序过于复杂和庞大。传统的异常处理机制无法保证程序的可靠运行, 而且采用判断或分支语句处理异常的方法不适合大量异常的处理, 更不能处理不可预知的异常。C++提供的异常处理机制逻辑结构非常清晰, 而且在一定程度上可以保证程序的健壮性。

10.1.2 异常处理的方法

C++处理异常的办法是: 如果在执行一个函数过程中出现异常, 可以不在本函数中立即处理, 而是发出一个信息, 传给它的上一级 (即调用函数) 来解决, 如果上一级函数也不能处理, 就再传给其上一级, 由其上一级处理。如此逐级上传, 如果到最高一级还无法处理, 运行系统一般会调用系统函数 `terminate`, 由它调用 `abort` 终止程序。

这样的异常处理方法使得异常的引发和处理机制分离, 而不是由同一个函数完成。这样做的好处是使底层函数(被调用函数)着重用于解决实际任务, 而不必过多地考虑对异常的处理, 以减轻底层函数的负担, 而把处理异常的任务上移到上层去处理。例如在主函数中调用十几个函数, 只需在主函数中设计针对不类型的异常处理, 而不必在每个函数中都设置异常处理, 这样可以大大提高效率。

C++处理异常的机制是由检查、抛出和捕获 3 部分组成, 分别由 3 种语句来完成: `try` (检查)、`throw` (抛出) 和 `catch` (捕获)。把需要检查的语句放在 `try` 中, `throw` 用来在出现异常时, 发出一个信息, 而 `catch` 用来捕获异常信息, 并在捕获到异常信息后对其进行处理。

1. 异常的抛出

抛出异常使用 `throw` 语句, 其格式如下:

`throw` 表达式;

如果在某段程序中发现了异常, 就可以使用 `throw` 语句抛出这个异常给调用者, 该异常由与之匹配的 `catch` 语句来捕获。`throw` 语句中的“表达式”是表示抛出的异常类型, 异常类

型由表达式的类型来表示。例如, 含有 throw 语句的函数 Div 可写成:

```
int Div(int x, int y)
{ if (y==0)
    throw y;      //抛出异常, 当除数 y 为 0 时, 语句 throw 将抛出 int 型异常
    return x/y;    //当除数 y 不为 0 时, 返回 x/y 的值
}
```

由于变量 y 的类型是 int, 所以当除数 y 为 0 时, 语句 throw 将抛出 int 型异常。

2. 异常的检查和捕获

异常的检查和捕获使用 try 语句和 catch 语句, 格式如式:

```
try
{
    被检查的复合语句
}
catch (异常类型声明 1)
{
    进行异常处理的复合语句 1
}
catch (异常类型声明 2)
{
    进行异常处理的复合语句 2
}
...
catch (异常类型声明 n)
{
    进行异常处理的复合语句 n
}
```

try 后的复合语句是被检查语句, 也是容易引起异常的语句, 这些语句称为代码的保护段。如果预料某段程序代码 (或对某个函数的调用) 有可能发生异常, 就将它放在 try 之后。如果这段代码 (或被调函数) 运行时真的遇到异常情况, 其中的 throw 表达式就会抛出这个异常。

catch 用来捕获 throw 抛出的异常, catch 子句后的复合语句是异常处理程序, 异常类型声明部分指明了 catch 子句处理的异常的类型。catch 在捕获到异常后, 由子句检查异常的类型, 即检查 throw 后表达式的数据类型与哪个 catch 子句的异常类型的声明一致, 如一致则执行相应的异常处理程序 (该子句后的复合语句)。例如, 用于处理除数为零异常的 try_catch 语句如下:

```
try                                     //检查异常
{
    cout<<"7/3"<<Div(7, 3)<<endl;      //被检查的复合语句
    cout<<"5/0"<<Div(5, 0)<<endl;
}
catch (int)                             //捕获异常, 异常类型是 int 型
{
    cout<<"除数为 0, 错误!"<<endl;     //进行异常处理的复合语句
}
```



```

}

```

处理除数为零异常的完整程序如例 10.2 所示。

例 10.2 处理除数为零异常的程序。

```

#include<iostream>
using namespace std;
int Div(int x, int y);           //函数 Div 的原型
int main()
{ try                           //检查异常
{ cout<<"7/3="<<Div(7, 3)<<endl;   //被检查的复合语句
  cout<<"5/0="<<Div(5, 0)<<endl;
}
catch (int)                     //捕获异常, 异常类型是 int 型
{ cout<<"除数为 0, 错误!"<<endl;   //进行异常处理的复合语句
}
cout<<"end"<<endl;
return 0;
}

int Div(int x, int y)
{ if (y==0)
    throw y;                    //抛出异常, 当除数 y 为 0 时, 语句 throw 将抛出 int 型异常
    return x/y;                //当除数 y 不为 0 时, 返回 x/y 的值
}

```

在主函数中, 首先执行 try 语句, 调用函数 Div(5, 0) 时发生异常, 由 Div 函数中语句 “throw y;” 抛出 int 型异常 (因为变量 y 是 int 类型), 被与之匹配的 catch 语句捕获 (因为两者的异常类型都是 int 型), 并在 catch 内进行异常处理后, 执行 catch 后面的语句。

程序运行结果如下:

```

7/3=2
除数为 0, 错误!    (异常处理)
end

```

在本例中, 进行异常处理的方法如下。

(1) 首先将需要检查的、也是容易引起异常的语句或程序段放在 try 块的花括号中。由于函数 Div 是可能出现异常的部分, 所以将以下语句放在 try 块中。

```

cout<<"7/3="<<Div(7, 3)<<endl;
cout<<"5/0="<<Div(5, 0)<<endl;

```

(2) 如果在执行 try 块内的复合语句过程中没有发生异常, 则 catch 子句不起作用, 流程转到 catch 子句后面的语句继续执行。

(3) 如果在执行 try 块内的复合语句 (或被调函数) 过程中发生异常, 则 throw 语句抛出一个异常信息。在本程序的中, 第 2 次执行函数 Div 时, 出现除数为零的异常, throw 抛出 int 类型的异常信息 y。throw 抛出异常信息后, 流程转到其上一级的函数 (即主函数 main)。因此不会执行函数 Div 中 if 语句之后的 return 语句。

(4) throw 抛出的异常信息传到 try_catch 结构, 系统寻找与之匹配的 catch 子句。本例中, y 是 int 型, 而 catch 子句的括号内指定的类型也是 int 型, 两者匹配, 即 catch 捕获了

该异常信息，就执行子句中的异常处理语句：

```
cout<<"除数为 0，错误！"<<endl;
```

(5) 执行异常处理语句后，程序继续执行 catch 子句后面的语句，在本程序中就执行语句：

```
cout<<"end"<<endl;
return 0;
```

说明：

(1) 被检测的语句或程序段必须放在 try 块中，否则不起作用。

(2) try 和 catch 块中必须有用花括号括起来的复合语句，即使花括号内只有一个语句也不能省略花括号。

(3) 一个 try_catch 结构中只能有一个 try 块，但却可以有多个 catch 块，以便与不同的异常信息匹配。catch 后面的括号中，一般只写异常信息的类型名。如例 10.3 所示程序。

例 10.3 有多个 catch 块的异常处理程序。

```
#include<iostream>
using namespace std;
int main()
{ double a=2.5;
  try                                //检查异常
  { throw a;                         //抛出异常
  }
  catch (int )                       //捕获异常，异常类型是 int 型
  { cout<<"异常发生！整数型！"<<endl; //进行异常处理的复合语句
  }
  catch (double )                   //捕获异常，异常类型是 double 型
  { cout<<"异常发生！双精度型！"<<endl; //进行异常处理的复合语句
  }
  cout<<"end"<<endl;
  return 0;
}
```

因为 a 定义为 double，所以“throw a;”抛出的异常类型为 double 型，被“catch (double)”捕获。程序运行结果如下：

```
异常发生！双精度型！
end
```

(4) 如果在 catch 子句中没有指定异常信息的类型，而用了删节号“...”，则表示它可以捕获任何类型的异常信息，例如：

例 10.4 有删节号“...”的异常处理程序。

```
#include<iostream>
using namespace std;
void func(int x)
{ if (x)
  { throw x;                        //抛出异常，语句 throw 抛出整型异常
  }
}
int main()
{ try                              //检查异常
{

```

```

func(5);
cout<<"No here!"<<endl;           //被检查的复合语句
}
catch (...)                        //捕获异常, 异常类型是任意类型
{ cout<<"任意类型异常!"<<endl;     //进行异常处理的复合语句
}
cout<<"end"<<endl;
return 0;
}

```

程序运行结果如下:

```
任意类型异常!
```

```
end
```

(5) 在某种情况下, 在 throw 语句中可以不包括表达式, 例如:

```
throw;
```

此时它将把当前正在处理的异常信息再次抛出, 给其上一层的 catch 块处理。

(6) C++中, 一旦抛出一个异常, 而程序又不捕获的话, 那么系统就会调用一个系统函数 terminate, 由它调用 abort 终止程序。

本节简单地介绍了异常处理的基本思想和方法, 更深入的了解, 请参阅有关专业资料。

10.2 命名空间和头文件命名规则

10.2.1 命名空间

一个大型软件通常是由多个模块组成的, 这些模块往往是由多人合作完成的, 不同的人分别完成不同的模块, 最后组合成一个完整的程序。假如不同的人分别定义了函数和类, 放在不同的头文件中, 在主文件需要用这些函数和类时, 就用#include 命令行将这些头文件包括进来。由于各头文件是由不同的人设计的, 有可能在不同的头文件中用了相同名字来定义的函数或类。这样在程序中就会出现命名冲突, 就会引起程序出错。另外, 如果在程序中用到第三方的库, 也容易产生同样的问题。为了解决这一问题, ANSI C++引入了命名空间, 用来处理程序中常见的同名冲突问题。所谓命名空间, 实际上就是一个由程序设计者命名的内存区域。程序设计者可以根据需要指定一些有名字的命名空间, 将各命名空间中声明的标识符与该命名空间标识符建立关联, 保证不同命名空间的名同标识符不发生冲突。声明命名空间的方法很简单, 下面的代码就是在命名空间 NS 中定义了两个简单变量 i 和 j:

```

namespace NS
{ int i=5;
  int j=10;
}

```

其中, namespace 是定义命名空间的所必须写的关键字, NS 是用户指定的命名空间的名字, 花括号内是命名空间的作用域。声明了命名空间后, 就可以解决名字冲突的问题。C++中命名空间的作用类似于操作系统中的目录和文件的关系, 由于文件很多, 不便管理, 而且容易重名, 于是人们设立若干子目录, 把文件分别放到不同的子目录中, 不同子目录中的文

件可以同名。调用文件时应指出文件路径。

除了用户可以声明自己的命名空间外，C++还定义了一个标准命名空间 `std`。在本书的各章节程序中，我们经常使用语句：

```
using namespace std;
```

其含义就是使用标准命名空间 `std`。

`std` (`standard` 的缩写) 是标准 C++ 指定的一个命名空间，标准 C++ 库中的所有标识符都是在这个名为 `std` 的命名空间中定义的，或者说标准头文件（如 `iostream`）中的函数、类、对象和类模板是在命名空间 `std` 中定义的。如果要使用输入输出流对象（如 `cin`、`cout`），就要告诉编译器该标识符可在命名空间 `std` 找到。其方法有两种，一种是像本书前面章节中所写的程序一样，在源文件中使用“`using namespace std;`”语句。例如：

```
#include<iostream>
using namespace std;
int main()
{ cout<<"Welcome to C++!"<<endl;
  return 0;
}
```

另一种方法是在该标识符前面加上命名空间及作用域运算符“`::`”。例如：

```
#include<iostream>
int main()
{ std::cout<<"Welcome to C++!"<<std::endl;
  return 0;
}
```

例 10.5 命名空间的使用举例 1。

```
#include<iostream>
namespace University          //声明命名空间，名为 University
{ int grade=3;
}
namespace Highschool          //声明命名空间，名为 Highschool
{ int grade=4;
}
int main()
{ std::cout<<"The university's grade is:"<<University::grade<<std::endl;
  std::cout<<"The highschool's grade is:"<<Highschool::grade<<std::endl;
  return 0;
}
```

在本例中声明了两个命名空间 `University` 和 `Highschool`，在各自的命名空间中都用到同名变量 `grade`，为了区分这两个 `grade` 变量，必须在其前面加上命名空间的名字和作用域运算符“`::`”。其中，“`University::grade`”为命名空间 `University` 中定义的 `grade`，“`Highschool::grade`”为命名空间 `Highschool` 中定义的 `grade`，“`std::cout`”为标准命名空间 `std` 中定义的流对象，“`std::endl`”为标准命名空间 `std` 中定义的操作符。

程序运行结果如下：

```
The university's grade is:3
```

```
The highschool's grade is:4
```

例 10.5 也可以改写成例 10.6 的形式。读者不难分析，这两种方法的运行结果是相同的。

例 10.6 命名空间的使用举例 2。

```
#include<iostream>
using namespace std;
namespace University          //声明命名空间, 名为 University
{ int grade=3;
}
namespace Highschool          //声明命名空间, 名为 Highschool
{ int grade=4;
}
int main()
{ cout<<"The university's grade is:"<<University::grade<<endl;
  cout<<"The highschool's grade is:"<<Highschool::grade<<endl;
  return 0;
}
```

说明:

当前使用的 C++ 库大多是几年前开发的, 由于 C++ 的早期版本中没有命名空间的概念, 库中的有关内容也没有放在 std 命名空间中, 因而在程序中不必对 std 进行声明。这也是目前的程序中没有使用“using namespace std;”语句的原因。但是, 用标准的 C++ 编程是应该对命名空间 std 的成员进行声明或限定的 (可以采用前面介绍过的任一种方法)。

10.2.2 头文件命名规则

由于 C++ 是从 C 语言发展而来的, 为了与 C 兼容, C++ 保留了 C 语言中的一些规定。例如, 在 C 语言中头文件用 .h 作为带后缀, 如 stdio.h、math.h 等。为了与 C 语言兼容, 许多 C++ 早期版本的编译系统头文件都是采用 “*.h” 形式, 如 iostream.h 等。但后来 ANSI C++ 建议头文件不带后缀 “.h”。近年推出的 C++ 编译系统新版本则采用了 C++ 的新方法, 头文件名不再有后缀 “.h”, 如 iostream、cmath 等。但为了使原来编写的 C++ 程序能够运行, 在 C++ 程序中使用头文件时, 既可以采用 C++ 中不带后缀的头文件, 也可以采用 C 语言中带后缀的头文件。

1. 带后缀的头文件的使用

在 C 语言程序中头文件包括后缀 .h, 如 stdio.h、string.h 等。由于 C 语言没有命名空间, 头文件不存放在命名空间中。因此在 C++ 程序中, 如果使用带后缀 “.h” 的头文件, 不必用命名空间。只需在文件中包含所用的头文件即可。例如:

```
#include<stdio.h>
```

2. 不带后缀的头文件的使用

C++ 标准要求系统提供的头文件不包括后缀 .h, 例如 string、iostream。为了表示 C++ 与 C 语言的头文件既有联系又有区别, C++ 所用的头文件不带后缀字符 .h, 而是在 C 语言的相应的头文件名之前加上前缀字符 c。例如, C 语言中的头文件 stdio.h, 在 C++ 中相应的头文件名为 cstdio。C 语言中的头文件 string.h, 在 C++ 中相应的头文件名为 cstring。

使用 C++ 中不带后缀的头文件时, 需要在程序中声明命名空间 std。例如:

```
#include<cstdio>          //相当于 C 程序中的#include<stdio.h>
#include<cstring>         //相当于 C 程序中的#include<string.h>
using namespace std;      //声明使用命名空间 std
```

使用头文件的两种方法是等价的，可以任意选用。但使用带后缀的头文件时，不需要在程序中声明命名空间 `std`。

10.3 应用举例

例 10.7 输入三角形的三边，求解三角形的面积。当输入边的长度小于或等于 0 时，抛出 `int` 型异常，给出警告并结束程序；当三条边的长度都大于 0，但不符合构成三角形条件（任意两边之和应大于第三边）时，抛出 `double` 型异常，给出警告并结束程序。

```
#include <iostream>
#include <cmath>
using namespace std;
double triangle(double a, double b, double c);           //计算三角形面积函数原型

int main()
{ double a, b, c;
  try                                                    //检查异常
  { cout<<"请输入三角形的三个边长(a、b、c)："<<endl;
    cout<<"a="; cin>>a;
    cout<<"b="; cin>>b;
    cout<<"c="; cin>>c;
    if (a<=0|| b<=0|| c<=0)
      throw 1;                                           //抛出 int 型异常

    while(a>0&&b>0&&c>0)
    { cout<<"a="<<a<<", b="<<b<<", c="<<c<<endl;
      cout<<"三角形的面积="<<triangle(a, b, c)<<endl;
      cout<<"请输入三角形的三个边长(a、b、c)："<<endl;
      cout<<"a="; cin>>a;
      cout<<"b="; cin>>b;
      cout<<"c="; cin>>c;
      if (a<=0|| b<=0|| c<=0)
        throw 1;                                         //抛出 int 型异常
    }
  }
  catch(double)                                          //捕获 double 型异常，并作相应处理
  { cout<<"这三条边不能构成三角形，异常发生，结束!"<<endl;
  }
  catch(int)                                             //捕获 int 型异常，并作相应处理
  { cout<<"a="<<a<<", b="<<b<<", c="<<c<<endl;
    cout<<"边长小于或等于 0，异常发生，结束!"<<endl; }
  return 0;
}

double triangle(double a, double b, double c)          //定义计算三角形面积函数
{ double s=(a+b+c)/2;
```

```

if(a+b<=c||b+c<=a||c+a<=b) throw 1.0;           //抛出 double 型异常
return sqrt(s*(s-a)*(s-b)*(s-c));
}

```

程序运行结果如下:

请输入三角形的三个边长(a、b、c):

a=3✓

b=5✓

c=6✓

a=3, b=5, c=6

三角形的面积=7.48331

请输入三角形的三个边长(a、b、c):

a=2✓

b=3✓

c=8✓

a=2, b=3, c=8

这 3 条边不能构成三角形, 异常发生, 结束!

实 验

实验目的和要求

1. 学习异常处理的声明和执行过程。
2. 学习命名空间的定义和使用方法。

实验内容和步骤

1. 分析并调试下列程序, 写出运行结果并分析原因。

```

//test10_1.cpp
#include<iostream>
using namespace std;
namespace NS1{
    int x=10;
}
namespace NS2{
    int x=20;
}
void main()
{ using NS1::x;
  using namespace NS2;
  cout<<"x="<<x<<endl;
}

```

2. 下面是一个用于处理文件打不开的异常处理程序, 分析程序并完成相应问题。

```

//test10_2.cpp
#include<fstream>
#include<iostream>
using namespace std;
int main()

```

```

{ ifstream source("file9_2.txt"); //打开文件
char line[128];
try{
    if(!source)
        throw "file9_2.txt"; //如果打开失败,抛出异常
}
catch(char*s)
{ cout<<"error opening the file "<<s<<endl;
  exit(1);
}
while(!source.eof()){ //判断是否到文件末尾
    source.getline(line, sizeof(line));
    cout<<line<<endl;
}
source.close();
return 0;
}

```

请回答以下问题:

- (1) 若其中没有 file9_2.txt 文件,则输出结果如何?
- (2) 在硬盘上建一个 file9_2.txt 文件,其文件内容自己定义。输出结果如何?
3. 编写一个程序,求输入数的平方根。设置异常处理,对输入负数的情况给出提示。

习 题

【10.1】什么叫异常处理?

【10.2】C++处理异常的办法是什么?有什么优点?

【10.3】什么是命名空间?

【10.4】C++处理异常的机制是由()三部分组成。

- A. 编辑、编译和运行 B. 检查、抛出和捕获
C. 编辑、编译和捕获 D. 检查、抛出和运行

【10.5】C++中实现异常处理的3种语句,除了 try 和 catch 外,还有()。

- A. throw B. class C. if D. return

【10.6】catch(...)一般放在其他 catch 子句的后面,该子句的作用是()。

- A. 抛出异常 B. 捕获所有类型的异常
C. 检测并处理异常 D. 有语法错误

【10.7】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
int f(int );
int main()
{ try
  { cout<<"4!="<<f(4)<<endl;
    cout<<"-2!="<<f(-2)<<endl;
  }
  catch (int n)

```



```

{ cout<<"n="<<n<<" 不能计算 n!."<<endl;
cout<<"程序执行结束."<<endl;
}
return 0;
}
int f(int n)
{ if (n<=0)
    throw n;
    int s=1;
    for (int i=1;i<=n;i++)
        s*=i;
    return s;
}

```

【10.8】写出下面程序的运行结果。

```

#include<iostream>
using namespace std;
namespace NS1
{ void fun()
  { cout<<"NS1:fun"<<endl; }
}
namespace NS2
{ void fun()
  { cout<<"NS2:fun"<<endl; }
}
int main()
{ NS1::fun();
  NS2::fun();
  return 0;
}

```

【10.9】设计一个程序，采用异常处理的方法，在输入学生类对象的数据时检验成绩输入是否正确。当输入的成绩大于 100 或小于 0 时抛出一个异常。

第 11 章

综合设计与实现

本章通过一个比较完整的程序,使读者在掌握 C++语言的基础上,进一步了解面向对象编程技术的优点,学会如何使用面向对象的方法来编写一个较大的应用程序。

11.1 需求分析

例 11.1 编写一个小型的学生信息管理系统,可以对中学生、大学生和研究生的信息进行简单的管理。每一类学生都包含有学生名、成绩 1、成绩 2、成绩 3 和平均成绩,其中平均成绩 = (成绩 1 + 成绩 2 + 成绩 3) / 3。每类学生还有区别于其他类学生的特殊信息,例如中学生有家长,大学生有专业,研究生有导师。要求通过本系统实现以下功能:

- (1) 输入学生的基本信息;
- (2) 根据学生名查询某个学生的信息;
- (3) 计算并显示某个学生的平均成绩。

11.2 系统分析

通过这个例子进一步说明了面向对象程序设计的基本方法,主要是类和对象,对象数组、静态成员、类的组合、继承、文件的输入输出等综合使用。

对于本系统中的 3 种不同种类的对象:中学生、大学生和研究生,抽取其共性特征形成一个基类:基本信息类 Record。然后在这个基类的基础上,分别派生出 3 个类:中学生类 Student、大学生类 U_student 和研究生类 Graduate。各类学生信息列表存放到文件中。

11.2.1 基本信息类的属性和操作

1. 基本信息类的属性

基本信息类的属性有:

学生类别编号、学生名、成绩 1、成绩 2、成绩 3、平均成绩。

为了方便信息的读取,程序中给每类学生设置了一个学生类别编号,以便区别各类学生。

2. 基本信息类的操作

基本信息类的操作有:

- 数据输入 输入各个学生对象的信息:学生名、成绩 1、成绩 2 和成绩 3;

- 数据输出 输出各个学生对象的信息：学生类别编号、学生名、成绩1、成绩2和成绩3；
- 计算平均成绩 由每个学生对象的成绩1、成绩2和成绩3计算出平均成绩。平均成绩的计算公式为（成绩1+成绩2+成绩3）/3。

11.2.2 各种学生类的属性和操作

各类学生继承基本信息类的共性特征，并增加了自己特有的属性。

1. 中学生类的属性和操作

中学生类的属性有：

（1）继承了基本信息类的属性：学生类别编号、学生名、成绩1、成绩2、成绩3和平均成绩；

（2）增加了中学生类区别于其他学生类的特殊属性：家长。

中学生类的操作有：

（1）数据输入：除了继承了基本信息类的数据输入功能外，增加了输入中学生类对象特殊属性“家长”信息的功能。

（2）数据输出：除了继承了基本信息类的数据输出功能外，增加了输出中学生类对象特殊属性“家长”信息的功能。

2. 大学生类的属性和操作

大学生类的属性有：

（1）继承了基本信息类的属性：学生类别编号、学生名、成绩1、成绩2、成绩3和平均成绩；

（2）增加了大学生类区别于其他学生类的特殊属性：专业。

大学生类的操作有：

（1）数据输入：除了继承了基本信息类的数据输入功能外，增加了输入大学生类对象特殊属性“专业”信息的功能。

（2）数据输出：除了继承了基本信息类的数据输出功能外，增加了输出大学生类对象特殊属性“专业”信息的功能。

3. 研究生类的属性和操作

研究生类的属性有：

（1）继承了基本信息类的属性：学生类别编号、学生名、成绩1、成绩2、成绩3和平均成绩；

（2）增加了研究生类区别于其他学生类的特殊属性：导师。

研究生类的操作有：

（1）数据输入：除了继承了基本信息类的数据输入功能外，增加了输入研究生类对象特殊属性“导师”信息的功能。

（2）数据输出：除了继承了基本信息类的数据输出功能外，增加了输出研究生类对象特殊属性“导师”信息的功能。

11.2.3 系统管理类的操作

系统的管理功能自成一个类：系统管理类。该类的主要操作有：

（1）输入学生的基本信息；

- (2) 根据学生名查询某个学生的信息;
- (3) 计算并显示某个学生的平均成绩。

11.3 系统设计

11.3.1 基类和派生类的设计

根据上面的分析,需要设计一个基类 Record(基本信息类)和它的三个派生类 Student(中学生类)、U_student(大学生类)和 Graduate(研究生类)。

基本信息类 Record 中的数据成员是 num(学生类别编号)、name(学生名)、score1(成绩1)、score2(成绩2)、score3(成绩3)、average(平均成绩)。

3个学生类除了继承基类 Record 的数据外,类 Student(中学生类)还增加了数据成员 patriarch(家长),类 U_student(大学生类)增加了数据成员 specialty(专业),类 Graduate(研究生类)数据成员增加了 mentor(导师)。

在基类中定义了构造函数和对所有类型学生的相同操作,成员函数 Get_num 负责取出学生类别编号,成员函数 Get_score1 负责取出成绩1,成员函数 Get_score2 负责取出成绩2,成员函数 Get_score3 负责取出成绩3,成员函数 Get_average 负责取出平均成绩,成员函数 Getname 负责取出学生名,成员函数 Compute_average 负责计算平均成绩,成员函数 Input 负责数据输入,成员函数 Output 负责数据输出。

系统管理类(System)的主要操作是:成员函数 In_information 负责输入学生信息,成员函数 Search 负责查询学生信息,成员函数 Out_average 负责计算并显示平均成绩,成员函数 Interface 负责界面的输出。

下面是各个类的定义。

```
//Record.h
class Record{                                //基本信息类
protected:
    int num;                                //学生类别编号
    char name[20];                          //学生名
    float score1;                          //成绩1
    float score2;                          //成绩2
    float score3;                          //成绩3
    float average;                        //平均成绩
public:
    Record(char* R_name=" ",float sco1=0,float sco2=0, float sco3=0);
    ~Record(){}
    int Get_num();                          //取学生类别编号
    float Get_score1();                    //取成绩1
    float Get_score2();                    //取成绩2
    float Get_score3();                    //取成绩3
    float Get_average();                  //取平均成绩
    char *Getname();                      //取学生名
```

```

        void Compute_average();           //计算平均成绩
        void Input();                     //数据输入
        void Output();                     //数据输出
};
class Student:public Record{             //中学生类
    char patriarch[20];                   //家长
public:
    Student(char* R_name=" ",float sco1=0,float sco2=0,
        float sco3=0,char *tea=" ");
    ~Student(){}
    void Input();
    void Output();
};
class U_student:public Record{            //大学生类
    char specialty[20];                   //专业
public:
    U_student(char* R_name=" ",float sco1=0,float sco2=0,
        float sco3=0,char *spe=" ");
    ~U_student(){}
    void Input();
    void Output();
};
class Graduate:public Record{             //研究生类
    char mentor[20];                      //导师
public:
    Graduate(char *R_name=" ",float sco1=0,
        float sco2=0,float sco3=0,char* men=" ");
    ~Graduate(){}
    void Input();
    void Output();
};
class System{                             //系统管理类
    Record A;
    Student B[10];
    U_student C[10];
    Graduate D[10];
    static int j1,j2,j3;
    void infor1();                         //输入中学生类对象数据
    void infor2();                         //输入大学生类对象数据
    void infor3();                         //输入研究生类对象数据
    void save();                           //将文件信息输出到内存
    void Search1(int h,char ch[20]);
    void Out_average1(int h,char* name);
    void Interface1();
public:
    System();
    void In_information();                 //输入学生信息
    void Search();                         //查询学生信息
    void Out_average();                   //计算并显示平均成绩
    void Interface();                     //界面输出
};

```

本例的对象模型图如图 11.1 所示。

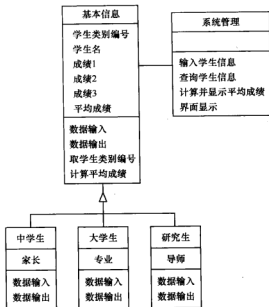


图 11.1 小型学生信息管理系统的对象模型图

11.3.2 系统管理类的设计

各个学生类涉及的操作都比较简单，具体的实现代码在本节后面给出。下面着重说明系统管理类 System 的操作设计。

1. 将数据文件中信息读入内存对象数组

当系统启动成功后，自动将学生信息从数据文件中读入内存各对象数组中。这个功能由系统管理类 System 的构造函数自动调用函数 save 完成的。例如：

```
system::system()
{ save(); }
```

所有学生的信息都存在一个数据文件中，由于不同种类学生信息的长度不相同，如何从数据文件里将这些数据读出并存放各自对应的对象数组中是一个问题，解决的方法是每次从数据文件中读取基类大小的一条信息存入基类对象中，并获得这条信息的种类编号，通过学生类别编号可以判定学生的类别，然后将指针回指到这条信息的开头，从学生文件中重新读取完整的学生信息，存入对应的对象数组中。例如：

基本信息类 Record 的对象是 A
中学生类 Student 的对象是 B[j1]

从数据文件里将相关的学生数据读出并存放对象 B[j1] 中，有关程序段如下：

```
fstream datafile(fileName, ios::out | ios::in | ios::binary);
datafile.read((char*) &A, sizeof(Record));
while(!datafile.eof())
{
```

```

a=A.Get_num();
switch(a){
case 1:
{
    datafile.seekp(-1* sizeof(class Record),ios::cur);
    datafile.read((char*)&B[jl],sizeof(Student));
    ...
}
...
}

```

2. 信息的输入

信息的输入功能由成员函数 `In_information` 来完成, 它根据要输入学生的类别分别调用对应的学生信息输入功能函数完成本类学生的输入。有 3 个类别的学生信息输入函数:

```

void infor1();           //输入中学生类对象数据
void infor2();           //输入大学生类对象数据
void infor3();           //输入研究生类对象数据

```

下面以 `infor1` 为例说明一条学生信息输入的实现过程:

```

void System::infor1()    //输入中学生类对象数据
{
    Student A;
    fstream datafile(fileName,ios::in|ios::out|ios::binary);
    datafile.seekp(0,ios::end);    //写指针指到文件尾部
    A.Input();
    datafile.write((char*)&A,sizeof(class Student));
    B[jl]=A;
    datafile.close();
}

```

函数 `Infor1` 完成一条中学生对象信息的输入。新增加的学生信息添加在文件的尾部, 基本信息类 `Record` 对象 `A` 的属性值(学生名等)的录入是调用中学生类 `Student` 提供的数据输入成员 `Input` 完成, 中学生类对象信息输入文件的同时, 也保存一份在内存中学生类对象数组 `B` 中。

3. 信息的查询

信息查询功能由成员函数 `Search` 来完成。它接收从键盘输入的学生类别编号和学生名, 在对应的对象数组中查找, 找到后通过调用对象的数据输出成员函数 `Output` 来显示学生信息。例如查询中学生信息的程序段如下:

```

if(strcmp(ch,B[s].Getname())==0)
{
    B[s].Output();
    ...
}

```

4. 平均成绩计算和显示

平均成绩的计算和显示也是按照学生名来进行。先接收从键盘输入的学生类别编号和学生名, 在对应的对象数组中查找, 找到后通过调用对象的平均成绩计算函数计算平均成绩并输出。例如计算并显示中学生平均成绩的程序段如下:

```

if(strcmp(name,B[s].Getname())==0)

```

```

{
    B[s].Compute_average();
    average=B[s].Get_average();
    cout<<"\t\t\t 学生名:"<<name<<endl;
    cout<<"\t\t\t 平均成绩:"<<average<<endl;
}

```

5. 界面设计及实现

程序运行主要由 System 类中的成员函数 Interface 来完成,通过选择不同的菜单项调用不同的成员函数完成各自的功能。例如程序运行的主界面如图 11.2 所示。

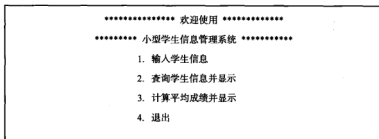


图 11.2 小型学生信息管理系统主界面

11.4 系统实现

完成前几个阶段的工作,现在就可以编码实现程序了。程序中包括了 3 个文件:Record.h、Record.cpp 和 main.cpp 文件。完整的程序及相应的说明如下:

```

//Record.h
class Record(                                //基本信息类
protected:
    int num;                                //学生类别编号
    char name[20];                          //学生名
    float score1;                            //成绩 1
    float score2;                            //成绩 2
    float score3;                            //成绩 3
    float average;                          //平均成绩
public:
    Record(char* R_name=" ",float sco1=0,float sco2=0, float sco3=0);
    ~Record(){}
    int Get_num();                          //取学生类别编号
    float Get_score1();                     //取成绩 1
    float Get_score2();                     //取成绩 2
    float Get_score3();                     //取成绩 3
    float Get_average();                    //取平均成绩
    char *Getname();                        //取学生名
    void Compute_average();                 //计算平均成绩
    void Input();                           //数据输入
    void Output();                          //数据输出
};

```



```

class Student:public Record{           //中学生类
    char patriarch[20];                //家长
public:
    Student(char* R_name=" ",float sco1=0,float sco2=0,
        float sco3=0,char *tea=" ");
    ~Student(){}
    void Input();
    void Output();
};

class U_student:public Record{         //大学生类
    char specialty[20];                //专业
public:
    U_student(char* R_name=" ",float sco1=0,float sco2=0,
        float sco3=0,char *spe=" ");
    ~U_student(){}
    void Input();
    void Output();
};

class Graduate:public Record{          //研究生类
    char mentor[20];                   //导师
public:
    Graduate(char *R_name=" ",float sco1=0,
        float sco2=0,float sco3=0,char* men=" ");
    ~Graduate(){}
    void Input();
    void Output();
};

class System{                          //系统管理类
    Record A;
    Student B[10];
    U_student C[10];
    Graduate D[10];
    static int j1,j2,j3;
    void infor1();                     //输入中学生类对象数据
    void infor2();                     //输入大学生类对象数据
    void infor3();                     //输入研究生类对象数据
    void save();                       //将文件信息输出到内存
    void Search1(int h,char ch[20]);
    void Out_average1(int h,char* name);
    void Interfacel();
public:
    System();
    void In_information();             //输入学生信息
    void Search();                     //查询学生信息
    void Out_average();                //计算并显示平均成绩
    void Interface();                  //界面显示
};

// Record.cpp
#include<iostream>
#include<string>
#include<fstream>
#include"Record.h"
using namespace std;
#define N 30                          //各类学生的最大数

```

```

char fileName[]="super.dat";           //存放学生信息的数据文件
Record::Record(char* R_name,float scol,float sco2,float sco3)
{
    strcpy(name,R_name);
    score1=scol;
    score2=sco2;
    score3=sco3;
}

int Record::Get_num()                   //取学生类别编号
{ return num; }

float Record::Get_score1()              //取成绩 1
{ return score1; }

float Record::Get_score2()              //取成绩 2
{ return score2; }

float Record::Get_score3()              //取成绩 3
{ return score3; }

float Record::Get_average()              //取平均成绩
{ return average; }

char*Record::Getname()                  //取学生名
{ return name; }

void Record::Compute_average()           //计算平均成绩
{ average=(score1+score2+score3)/3; }

void Record::Input()                    //数据输入
{ cout<<"\t\t 学生名:";
  cin>>name;
  cout<<"\t\t 成绩 1:";
  cin>>score1;
  cout<<"\t\t 成绩 2:";
  cin>>score2;
  cout<<"\t\t 成绩 3:";
  cin>>score3;
}

void Record::Output()                   //数据输出
{ cout<<endl;
  cout<<"\t\t 所要查看的学生信息:"<<endl;
  cout<<"\t\t 学生类别号:"<<num<<endl;
  cout<<"\t\t 学生名: " <<name<<endl;
  cout<<"\t\t 成绩 1: " <<score1<<endl;
  cout<<"\t\t 成绩 2: " <<score2<<endl;
  cout<<"\t\t 成绩 3: " <<score3<<endl;
}

Student::Student(char* R_name,float scol,
    float sco2,float sco3,char*tea):Record(R_name,scol,sco2,sco3)
{ num=1;
  strcpy(patriarch,tea);
}

void Student::Input()
{ Record::Input();
  cout<<"\t\t 家长:";
  cin>>patriarch;
}

void Student::Output()
{ Record::Output();
  cout<<"\t\t 家长:"<<patriarch<<endl;
}

```

[Home](#)
[About Us](#)
[Contact Us](#)
[Privacy Policy](#)
[Terms of Service](#)
[FAQ](#)

```

        break;
    case 2:                                     //输入大学生类对象数据
        infor2();
        break;
    case 3:                                     //输入研究生类对象数据
        infor3();
        break;
    case 4:                                     //界面输出
        Interface();
        break;
    default:
        cout<<"\t\t\t\t 没有此类学生!"<<endl;
        continue;
    }
    cout<<"\t\t\t\t 信息存储成功! "<<endl;
    cout<<"\t\t\t\t 是否继续输入(y/n)?"<<endl;
    cin>>t;
    cout<<endl;
    if(!(t=='Y' || t=='y'))
        again=0;
    }
    Interface();                               //界面输出
}

void System::infor1()                          //输入中学生类对象数据
{
    Student A;
    fstream datafile(fileName, ios::in|ios::out|ios::binary);
    datafile.seekp(0, ios::end);              //写指针指到文件尾部
    A.Input();
    datafile.write((char*)&A, sizeof(class Student));
    B[j1]=A;
    datafile.close();
}

void System::infor2()                          //输入大学生类对象数据
{
    U_student A;
    fstream datafile(fileName, ios::in|ios::out|ios::binary);
    datafile.seekp(0, ios::end);
    A.Input();
    datafile.write((char*)&A, sizeof(class U_student));
    C[j2]=A;
    datafile.close();
}

void System::infor3()                          //输入研究生类对象数据
{
    Graduate A;
    fstream datafile(fileName, ios::in|ios::out|ios::binary);
    datafile.seekp(0, ios::end);
    A.Input();
    datafile.write((char*)&A, sizeof(class Graduate));
    D[j3]=A;
    datafile.close();
}

void System::save()                            //将文件信息输出到内存
{
    int a;
    fstream datafile(fileName, ios::out|ios::in|ios::binary);
    datafile.read((char*)&A, sizeof(Record));
}

```

```

while(!datafile.eof())
{
    a=A.Get_num();
    switch(a)
    { case 1:
      { datafile.seekp(-1* sizeof(class Record),ios::cur);
        datafile.read((char*)&B[j1],sizeof(Student));
        j1++;
        break;
      }
      case 2:
      { datafile.seekp(-1*sizeof(class Record),ios::cur);
        datafile.read((char*)&C[j2],sizeof(U_student));
        j2++;
        break;
      }
      case 3:
      { datafile.seekp(-1*sizeof(class Record),ios::cur);
        datafile.read((char*)&D[j3],sizeof(Graduate));
        j3++;
        break;
      }
      default:
        break;
    }
    datafile.read((char*)&A,sizeof(Record));
}
datafile.close();
}

void System::Search1(int h,char ch[20])
{ int s=0,found=0;
  switch(h)
  { case 1:
    while(s<N)
    { if(strcmp(ch,B[s].Getname())==0)
      { B[s].Output();
        cout<<"\t\t\t*****"<<endl;
        found=1;
      }
      s++;
    }
    break;
  case 2:
    while(s<N)
    { if(strcmp(ch,C[s].Getname())==0)
      { C[s].Output();
        cout<<"\t\t\t*****"<<endl;
        found=1;
      }
      s++;
    }
    break;
  case 3:
    while(s<N)

```

```

        { if(strcmp(ch,D[s].Getname())==0)
          { D[s].Output();
            cout<<"\t\t\t*****"<<endl;
            found=1;
          }
          s++;
        }
        break;
    }
    if(found==0)
        cout<<"\n\n\t\t\t 对不起，该类别中没有您所查询的学生!"<<endl;
}

void System::Search()                                //查询学生信息
{ int rev;
  char name[20];
  int again=1;
  char t;
  while(again)
  { Interface();
    cin>>rev;
    cout<<"\t\t\t 请输入要查询的学生名:";
    cin>>name;
    Search1(rev,name);
    cout<<"\t\t\t\t 是否继续查询(y/n)?";
    cin>>t;
    cout<<endl;
    if( !(t=='Y' || t=='y'))
        again=0;
  }
  Interface();
}

void System::Out_average1(int h,char* name)          //计算并显示平均成绩
{ int s=0,found=0;
  float average;
  switch(h)
  { case 1:
    while(s<N)
    { if(strcmp(name,B[s].Getname())==0)
      { B[s].Compute_average();
        average=B[s].Get_average();
        found=1;
      }
      s++;
    }
    break;
  case 2:
    while(s<N)
    { if(strcmp(name,C[s].Getname())==0)
      { C[s].Compute_average();
        average=C[s].Get_average();
        found=1;
      }
      s++;
    }
  }
}

```

```

    }
    break;
case 3:
    while(s<N)
    { if(strcmp(name,D[s].Getname())==0)
      { D[s].Compute_average();
        average=D[s].Get_average();
        found=1;
      }
      s++;
    }
    break;
}
if(found==0)
    cout<<"\n\n\t\t 对不起,该类别中没有您所查看平均成绩的学生!"
    <<endl;
else
{ cout<<"\t\t\t 学生名:"<<name<<endl;
  cout<<"\t\t\t 平均成绩:"<<average<<endl;
  cout<<"\t\t\t ***** "<<endl;
}
}

void System::Out_average() //计算并显示平均成绩
{ int rev;
  char name[20];
  int again=1;
  char t;
  while(again)
  { Interface();
    cout<<"\n\t\t\t 请输入所要查看平均成绩的学生类别:";
    cin>>rev;
    cout<<"\n\t\t\t 请输入所要查看平均成绩的学生名:";
    cin>>name;
    Out_average1(rev,name);
    cout<<"\t\t\t\t 是否继续查看平均成绩(y/n)?";
    cin>>t;
    cout<<endl;
    if(!(t=='Y' || t=='y'))
        again=0;
  }
  Interface();
}

void System::Interface() //界面输出
{ int rev;
  cout<<"\n\n\n\n\n\n\n";
  cout<<"\t\t\t *****欢迎使用";
  cout<<"*****"<<endl;
  cout<<"\t\t\t *****小型学生信息管理系统";
  cout<<"*****"<<endl;
  cout<<"\t\t\t 1. 输入学生信息 "<<endl;
  cout<<"\t\t\t 2. 查询学生信息并显示 "<<endl;
  cout<<"\t\t\t 3. 计算平均成绩并显示 "<<endl;
  cout<<"\t\t\t 4. 退出 "<<endl;
}

```

```

        cout<<"\t\t 请您选择(1-4): ";
        cin>>rev;
        switch(rev)
        { case 1:
            In_information();
            break;
          case 2:
            Search();
            break;
          case 3:
            Out_average();
            break;
          case 4:
            exit(0);
        }
    }
//main.cpp
#include "Record.h"
int main(void)
{ System s;
  s.Interface();
  return 0;
}

```

图 11.3 和图 11.4 为程序运行后的部分界面。



图 11.3 输入学生信息界面

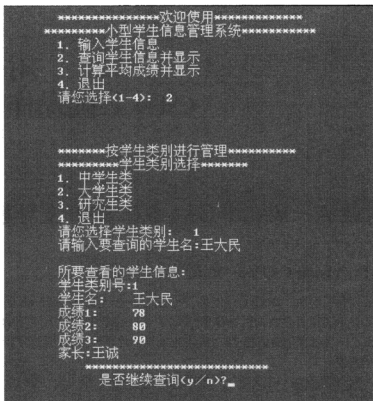


图 11.4 查询学生信息界面

实 验

实验目的和要求

学习使用面向对象的方法来编写一个较大的应用程序

实验内容和步骤

【11.1】按习题 11.1 的要求编写一个程序，在计算机上编辑、编译、调试和运行这个程序，并写出程序的运行结果。

习 题

【11.1】设计一个程序，用于管理某高校教师、一般员工和学生的数据。

学生的数据包括姓名、年龄、身份证号、年级和学费；

教师的数据包括姓名、年龄、身份证号、课时和（每小时）课时费；

一般员工的数据包括姓名、年龄、身份证号和月薪。

附录 C++上机操作介绍

附录 A Visual C++ 6.0 的开发环境

附录 A.1 Visual C++ 6.0 集成开发环境概述

自从 Microsoft 公司发布 Visual C++ 以来, Visual C++ 已经成为 Windows 操作系统环境下最主要的应用系统开发工具之一, 是目前用得最多的 C++ 编译系统, 现在常用的是 Visual C++ 6.0 版本。Visual C++ 是可视化的编程工具, 其友好的操作界面和强大的程序编译与调试工具, 都深受程序员的喜爱。

Visual C++ 6.0 有英文版和中文版, 两者的操作方法基本是相同的, 用户可根据自身的情况予以选用。为了方便读者, 本书尽可能将两种版本的操作方法同时介绍给大家。

如果你使用的计算机上已经安装了 Visual C++ 6.0, 使用时只需单击 Windows 操作系统桌面上的“开始”→“程序”→“Microsoft Visual Studio 6.0”→“Visual C++ 6.0”命令, 就会出现如图 A-1 所示的 Visual C++ 6.0 的主窗口。

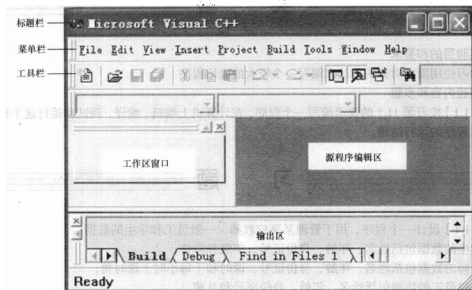


图 A-1

主窗口由标题栏、菜单栏、工具栏、工作区窗口、源程序编辑窗口和输出窗口组成。

主窗口最上方是标题栏，显示所打开的应用程序名。标题栏左端是控制菜单图标，单击后弹出窗口的控制菜单。标题栏右端从左至右有 3 个控制按钮，分别为最小化、最大化（还原）和关闭按钮，可用它们快速设置窗口的大小。

标题栏下方是菜单栏，由 File（文件）、Edit（编辑）、View（查看）、Insert（插入）、Project（工程）、Build（编译）、Tools（工具）、Window（窗口）和 Help（帮助）9 个主菜单项组成，其中每个主菜单项又由多个菜单项和子菜单组成。以上各菜单项后括号中的内容是 Visual C++ 6.0 中文版中的中文显示，以下同。

除了菜单外，在 Visual C++ 开发环境窗口的不同地方右击鼠标还可以弹出相应的快捷菜单。这些菜单的功能需要读者自己去实践，在此不再介绍。

为了用户操作方便，Visual C++ 系统在主窗口中提供了多种工具栏，每种工具栏中有若干个按钮，每个工具栏中的按钮表示某种操作（如新建一个文本窗口、保存当前文件等）。在鼠标指向某个按钮时，将显示出该按钮的功能。

工具栏的下方有两个窗口：左窗口是项目工作区窗口（简称工作区窗口），右窗口是源程序编辑窗口。工作区窗口用来显示所设定的工作区的相关信息，源程序编辑窗口用来输入和编辑源程序。在 Visual C++ 中可同时打开多个源程序编辑窗口，源程序编辑窗口将以平铺或重叠的方式显示。

在项目工作区窗口和源程序编辑窗口的下方是一个输出窗口，当编译、连接时输出窗口会显示编译和连接信息。

工作区窗口可通过工具栏中 Workspace 按钮隐藏或显示；输出窗口可通过单击工具栏中 Output 按钮隐藏或显示。隐藏这些窗口可以扩大源程序编辑窗口的大小。

附录 A.2 常用功能键及其意义

为了使程序员能够方便快捷地完成程序开发，Visual C++ 开发环境提供了大量的功能键（快捷键）来简化一些常用操作的步骤。功能键操作直接、简单，而且非常方便，使程序员能够方便快捷地完成程序开发。表 A-1 列出了一些最常用的功能键，读者可以在实验过程中逐步掌握。

表 A-1 常用功能键表

操作类型	功能键	对应菜单命令	功 能
File（文件）操作	Ctrl+N	File→New	创建新的文件、项目或工作区
	Ctrl+O	File→Open	打开已存在的文件、项目等
	Ctrl+S	File→Save	将当前文件保存到磁盘
Edit（编辑）操作	Ctrl+Z	Edit→Undo	取消最近一次的编辑操作
	Ctrl+Y	Edit→Redo	恢复到使用 Undo 命令前的编辑效果
	Ctrl+X	Edit→Cut	删除选定的文本，同时将其复制到剪贴板
	Ctrl+C	Edit→Copy	将选定的文本复制到剪贴板
	Ctrl+V	Edit→Paste	将剪贴板中的内容插入到当前光标处
	Ctrl+A	Edit→Select All	用于选定当前源程序编辑窗口中的所有内容
	Ctrl+F	Edit→Find	用于查找指定的字符串

续表

操作类型	功能键	对应菜单命令	功 能
Edit (编辑) 操作	Ctrl+H	Edit→Replace	用于替换指定的字符串
	Del	Edit→Del	删除当前选定的文本, 如果没有选定文本, 则删除光标后面的一个字符
Build (编译) 程序操作	Ctrl+F7	Build→Compiler xxx.cpp	编译当前源文件
	Ctrl+F5	Build→Run xxx.exe	运行当前项目
	F7	Build→Build xxx.exe	建立可执行程序
	F5	Build→Start Debugging	启动调试程序
Debug (调试) 操作	F5	Debug→Go	如果源代码中设置了断点, 则从断点处运行到下一个断点处; 若未设置断点, 则运行程序到结束
	F11	Debug→Step into	单步执行下一条语句, 如果该语句中含有函数调用, 则会转入函数体内部进行单步执行
	F10	Debug→Step over	单步执行下一条语句, 如果该语句中含有函数调用, 则会将其作为一个整体执行, 而不会转入函数体内部进行单步执行
	shift-F11	Debug→Step out	跳出一个函数体的内部, 而继续进行单步执行
	F9		设置/清除断点
	Ctrl+F10	Debug→Run to cursor	进入调试状态, 并使程序运行到光标所在的位置
	shift+F9	Debug→QuickWatch	快速查看和修改变量或表达式的值
	Shift+F5	Debug→Stop Debugging	停止调试过程并返回到程序编辑状态

注: 当 VC++ 6.0 集成开发环境进入调试状态时, “Build” 菜单会被 “Debug” 菜单所取代。

附录 B 建立和运行单文件程序

C++程序可分为单文件程序和多文件程序。单文件程序是指一个程序只由一个源文件组成，在初学 C++语言时，大多数情况下编写的程序是单文件程序。我们先通过一个简单的例子，介绍建立和运行单文件程序的方法，下一节再介绍多文件程序的建立和运行的方法。

附录 B.1 编辑 C++源程序

1. 编辑一个新的 C++源程序

启动 Visual C++ 6.0 编译系统后，出现如图 A-1 所示的 Visual C++ 6.0 的主窗口。在主菜单栏中选择“File（文件）”命令，出现一个下拉式菜单，再选择该菜单中的“New（新建）”命令（见图 B-1），也可以用功能键 Ctrl+N（以下介绍中，凡菜单命令有相应的功能键的都可使用功能键操作），这时又出现一个“New（新建）”对话框。该框中又有 4 个标签项和选择框，选择“Files（文件）”标签，在它的下拉式菜单中，选择“C++ Source File”选项，表示要建立一个新的 C++源程序文件。然后在选择框的右半部分的 Location（目录）文本框中输入准备编辑的源文件的存储路径（现假设为 D:\C++），表示准备编辑的源程序文件将存放在 D:\C++子目录下。在其上方的“File（文件）”文本框中输入准备编辑的源程序文件的名字（假如现在输入 lab1_1.cpp），如图 B-2 所示。这样，就将进行输入和编辑的源程序以 lab1_1.cpp 为文件名存放在 D 盘的 C++目录下。

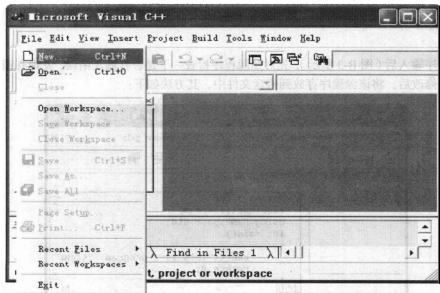


图 B-1

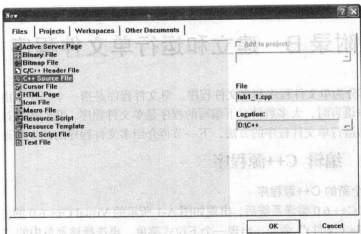


图 B-2

单击“OK（确定）”按钮后，回到 Visual C++主窗口，可以看到光标在程序编辑窗口中闪烁，表示源程序编辑窗口已激活，可以输入和编辑源程序了。

通过键盘输入以下源程序：

```
//lab1_1.cpp
#include<iostream>
using namespace std;
int main()
{
    cout<<"This is a program. "<<endl;
    return 0;
}
```

源程序输入后（图 B-3 所示），先对照源程序检查一下是否有输入错误，发现有错误就进行修改。修改后，将该源程序存放到磁盘文件中，其方法如下：

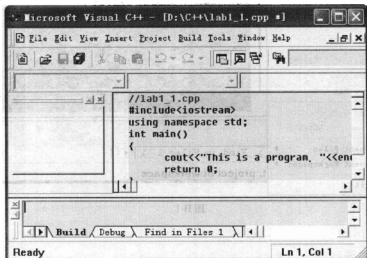


图 B-3

在主菜单中选择“File (文件)”命令,在其下拉式菜单中选择“Save (保存)”命令即可,如图 B-4 所示。

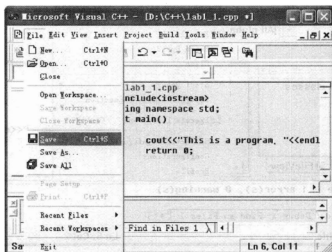


图 B-4

如果不想将源程序存放在原先指定的文件中,可以选择“Save As (另存为)”命令,屏幕上出现“Save As”对话框,在该对话框中,键入指定的文件路径和文件名。接着,按“Save (存入)”按钮或在键入的文件名后按回车键,就完成了该程序的存盘工作。

2. 打开一个已有的 C++源程序

如果你想编辑的源程序已经存在,则可通过以下方法将其打开:

(1) 在“我的电脑”或“资源管理器”中按路径找到已有的 C++程序名(如 lab1_1.cpp)。
 (2) 双击此文件名,则进入了 VC++集成环境,并打开了该文件,程序已显示在源程序编辑窗口中,如同图 B-3 所示。或者在 VC++主窗口中按“File (文件)→Open (打开)”命令,在弹出的“Open (打开)”对话框中,选择要打开的文件后,按“Open (打开)”按钮后,予以打开。

(3) 若不需修改,则可直接进行编译、连接和运行(方法见以下章节)。

(4) 若需修改,则修改后进行编译、连接和运行,修改后的程序自动保存在原来的文件中。当然也可以在编译前选择“File (文件)→Save (保存)”命令保存该文件;若不想将修改后的程序存放在原先的文件中,也可以选择“File (文件)→Save As (另存为)”命令,将它以另一个文件名(例如 test1.cpp)存放。

附录 B.2 编译和连接 C++程序

1. 程序的编译

选择菜单项“Build (编译)”→“Build (编译)”的下拉式菜单,在该下拉式菜单中选择“Compile lab1_1.cpp (编译 lab1_1.cpp)”命令后,系统将对当前的源程序文件 lab1_1.cpp 进行编译,如图 B-5 所示。

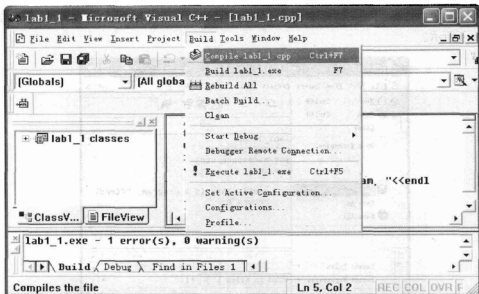


图 B-5

选择“Compile lab1_1.cpp (编译 lab1_1.cpp)”命令后，屏幕上出现一个对话框，内容是“This build command requires an active project workspace.Would you like to create a default project workspace?”（此编译命令要求有一个活动的项目工作区，你是否同意建立一个默认的项目工作区？），单击 Yes（是）按钮，表示同意后，开始编译。在编译过程中，编译系统检查源程序中是否有语法错误，将所发现的错误显示在屏幕下方的输出窗口中，如图 B-6 所示。

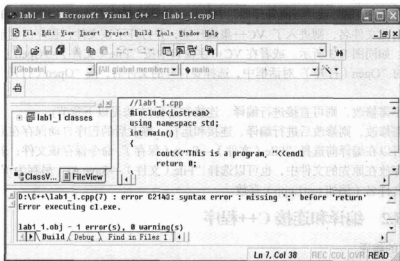


图 B-6

2. 程序的简单调试

一个程序，特别是大型程序，编写完成后往往会存在这样或那样的错误。有些错误在编

译阶段由编译系统指出，称为语法错误。这其中包括：

- 未定义的标识符（如函数名、变量名、类等）；
- 数据类型、参数类型及个数不匹配；
- 其他的语法错误。

在 Visual C++ 中，将这些语法错误分为三类：fatal error（致命错误）、error（错误）和 warning（警告）。

如果程序中有 fatal error 类型的错误，编译将会立即停止，你必须采取措施改正并重新启动编译。致命错误应是很少的。

如果程序中有 error 类型的语法错误，就不能通过编译，也就无法形成目标程序，当然也不能运行了。

如果程序中有 warning 类型的语法错误，这是一种轻微的语法错误，不影响其生成目标程序和可执行程序，但有可能影响运行的结果，所以也要将其修正。

在编译过程中，编译系统检查源程序中是否有语法错误，将所发现的错误显示在屏幕下方的输出窗口中。每个错误都给出其所在的文件名、行号、错误的类型和编号，以及错误的原因。在图 B-6 的输出窗口中可以看到编译的信息，指出此源程序有一个 error 类型的语法错误，错误的原因是在第 7 行“return”语句之前缺少一个“;”号。在输出窗口中双击这个错误，该错误将被高亮显示，在状态栏上显示出错内容，并定位到相应的程序行中，且该程序行的最前面有个蓝色箭头标志，如图 B-7 所示。若有多个错误，按 F4 键可显示下一个错误，并定位到相应的源程序行。

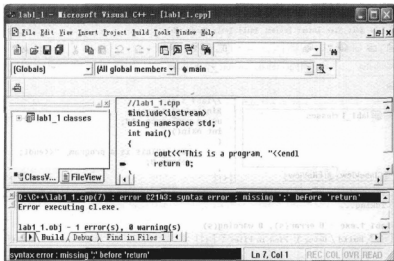


图 B-7

经检查，在源程序的第 6 行 cout 语句的末尾缺少一个“;”号。加上了“;”号后，再重新编译，如果有错误，再继续修改，直到使程序既无 error 错误，又无 warning 错误为止。此时编译信息提示：“lab1_1.obj - 0 error(s), 0 warning(s)”，也就是说，既没有 error 类型的语法错误，也没有 warning 类型的语法错误，这时将产生一个 lab1_1.obj 文件，如图 B-8 所示。

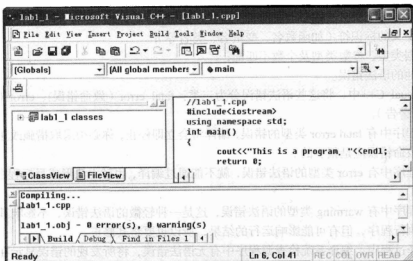


图 B-8

3. 程序的连接

编译无错后, 再进行连接, 这时选择“Build (编译)”菜单中的“Build lab1_1.exe (构建 lab1_1.exe)”命令。表示要求连接并建立一个可执行文件 lab1_1.exe。这时在输出窗口中显示连接的信息, 说明没有发现错误, 生成了一个可执行文件 lab1_1.exe 如图 B-9 所示。

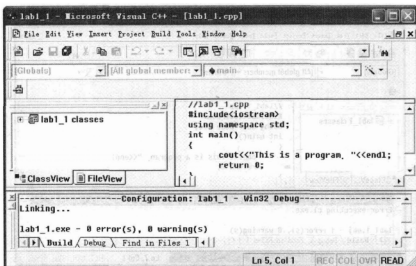


图 B-9

以上介绍的是分步进行程序的编译和连接, 实际上对有经验的用户来说也可直接选择“Build->Build lab1_1.exe”命令一次完成编译和连接。

附录 B.3 程序的运行

运行可执行文件的方法是选择“Build (编译)”菜单项中“Execute lab1_1.exe (执行

Execute lab1_1.exe)”命令，则可运行该可执行文件，如图 B-10 所示。

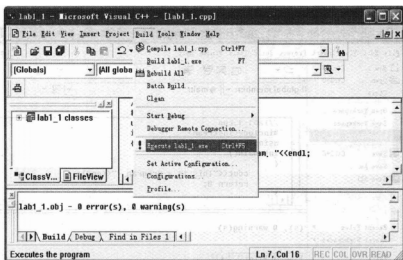


图 B-10

程序执行后，运行结果将显示在另外一个专门用来显示输出结果的窗口中，如图 B-11 所示。



图 B-11

其中，“This is a program.”是程序运行的结果。“Press any key to continue”是系统自动加上的一行信息，告诉用户“按任何一键就可继续”。此时，按任意键后，将返回到 Visual C++ 的主窗口。

附录 B.4 关闭工作区

在以上建立和运行单文件时，没有建立工作区，也没有建立项目文件，而是直接建立源程序。这主要考虑到读者大多是初学者，应尽量简化手续。实际上，在编译每一个程序时都需要建立一个工作区，如果用户未指定，系统会自动建立工作区，并赋予它一个默认名（此时以文件名作为工作区名）。

在执行完一个程序，编辑和运行新的程序前，应正确地使用关闭工作区命令来终止这个程序，安全地保护好这个程序。执行“File（文件）->Close Workspace（关闭工作区）”命令，

则结束对该程序的操作(见图 B-12 所示),接着,就可以编辑和运行新程序了。若要退出 VC++ 环境,则执行“File(文件)→Exit(退出)”命令。

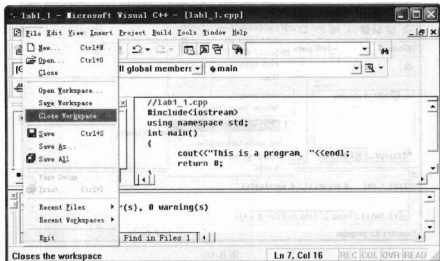


图 B-12

附录 C 建立和运行多文件程序

以上介绍的是单文件程序的建立和运行。本节介绍多文件程序建立和运行的方法。多文件程序是指一个程序中至少包含两个文件。如果一个程序包含多个源程序文件，则需要建立一个项目文件（project file），在这个项目文件中包含多个文件（包括源文件和头文件）。

使用 Visual C++ 6.0 建立和运行多文件程序的大体步骤如下。

- （1）编辑程序中需要的多个文件。
- （2）创建一个项目工作区(Workspace)。用户也可以不建立项目工作区，而由系统在建立项目文件时自动建立项目工作区，这个自动建立的项目工作区的名字与项目文件名相同。
- （3）创建项目文件(Project file)。
- （4）将多个文件添加到项目文件中。
- （5）编辑和连接项目文件。
- （6）运行项目可执行文件。

附录 C.1 编辑程序中需要的多个文件

编辑程序中多个文件的方法与前面讲述的编辑单文件的方法相同。在主菜单栏中选择“File（文件）”命令，出现一个下拉式菜单，再选择该菜单中的“New（新建）”命令，出现一个“New（新建）”对话框。在该对话框中，选择“Files（文件）”标签，在它的下拉式菜单中，选择“C++ Source File”选项。然后在选择框的 Location（目录）文本框中输入源文件的存储路径（例如 D:\C++），在“File（文件）”文本框中输入准备编辑的源程序文件的名字（假如输入 file1.cpp）。单击“OK（确定）”按钮后，回到 Visual C++ 主窗口，可以看到光标在源程序编辑窗口中闪烁，此时输入其中的一个文件。用同样的方法再输入该程序的其他文件。每输入一个文件，修改无错后，单击“File（文件）”菜单项的下拉式菜单中的“Save（保存）”或“Save As（另存为）”命令，分别将输入的文件按指定的文件名存好。

假设现在要编辑的程序由以下两个文件组成。

```
file1.cpp 文件:
#include<iostream>
using namespace std;
int add(int , int, int);
int main()
{ int a, b, c;
  a=1;
  b=2;
  c=3;
  cout<<add(a, b, c)<<endl;
  return 0;
}
file2.cpp 文件:
int add(int x, int y, int z)
{
```

```

    return x+y+z;
}

```

将上述两个文件按上述方法输入和编辑后，存放在文件路径“D:\C++”中。

附录 C.2 创建项目文件

要建立和运行多文件程序，可以先建立项目工作区，然后再建立项目文件，但这种方法步骤比较多。考虑到读者多是初学者，在此介绍一种简化的办法，即用户只建立项目文件，不建立项目工作区，而由系统自动建立项目工作区。

创建一个空的项目文件，用来存放该程序的上述两个文件。创建一个空项目文件的方法如下：

(1) 单击“File (文件)”菜单项的下拉式菜单中的“New (新建)”命令，出现“New (新建)”对话框，选择该对话框中的“Projects (中文版显示为:工程)”标签。在该标签的对话框中，单击“Win32 Console Application”选项。

(2) 在 Projects 标签对话框的右侧“Project name (中文版显示为:工程)”文本框内输入一个项目文件名，例如，输入指定的项目文件名 pro1，然后回车。此时，在“Location”文本框内生成一个路径名，该路径名可以修改（假如修改为 D:\C++），此时可以看到：在右部的中间单选按钮处默认选定了“Create new workspace (创建新工作区)”，这是由于用户未指定工作区，系统会自动开辟一个新工作区（名字与项目文件名相同）如图 C-1 所示。

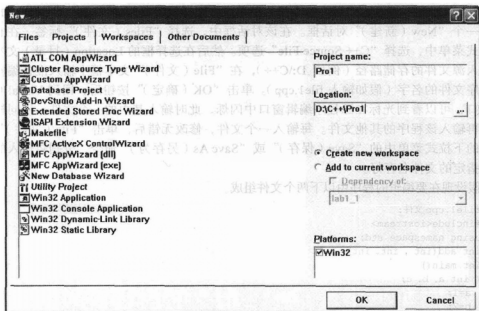


图 C-1

(3) 单击对话框的 OK 按钮。这时，屏幕上出现“Win32 Console Application-Step1 of 1”对话框，如图 C-2 所示。

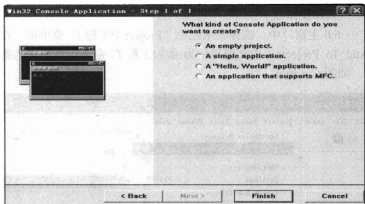


图 C-2

该对话框上方出现提示信息：“What kind of Console Application do you want to create? (请选择你所要创建的控制台应用程序的类型?)”，这时选择“An empty project”选项，单击该对话框下方的“Finish (完成)”按钮。

这时，屏幕上出现“New Project Information (新建工程信息)”对话框，该对话框告诉用户所创建的控制台应用程序新框架项目的特性。单击该对话框下方的“OK (确定)”按钮，返回到 Visual C++ 6.0 主窗口。

(4) 在主窗口的如左部窗口下方单击“File View”按钮，窗口中显示“Workspace 'pro1' 1 project(s)”，如图 C-3 所示。说明系统已自动建立了一个工作区，由于用户未指定工作区，系统将项目文件名 pro1 同时作为工作区名。

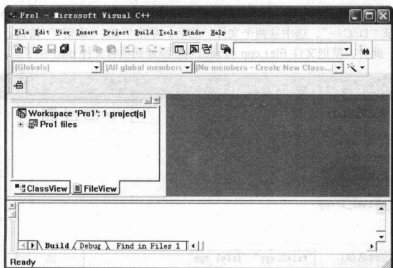


图 C-3

附录 C.3 将多个文件添加到项目文件中去

创建了一个空的项目文件 pro1 后，需要将事先编辑好的 file1.cpp 和 file2.cpp 文件添加到

项目文件 pro1 中。具体操作如下：

在 Visual C++ 6.0 主窗口中，选择菜单栏中“Project（工程）”菜单项，在出现的下拉式菜单中单击“Add To Projects（中文版显示为：添加工程）”命令，在弹出的级联菜单中单击“Files...”命令，如图 C-4 所示。

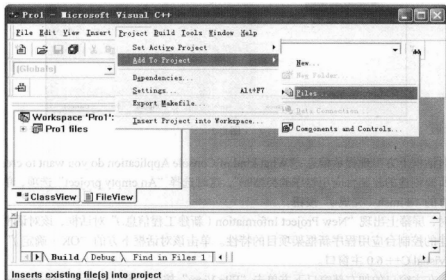


图 C-4

在选择“Files...”命令后，弹出“Insert Files into Project”对话框，如图 C-5 所示。在该对话框中，先确定项目文件 pro1，显示在“Insert into(插入到)”框内。打开 file1.cpp 和 file2.cpp 所在的子目录“D:\C++”，选中这两个文件。然后，单击“OK（确定）”按钮，则完成添加文件的任务。此时，就把文件 file1.cpp 和 file2.cpp 添加到项目文件 pro1 中了。

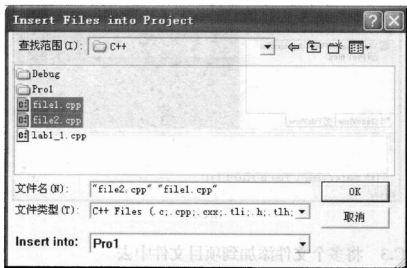


图 C-5

附录 C.4 编译和连接项目文件

由于文件 file1.cpp 和 file2.cpp 已被添加到项目文件 pro1 中了。因此只需对项目文件 pro1 进行统一的编译和连接。选择菜单栏中“Build (编译)”菜单项的下拉式菜单中的“Build pro1.exe (构建 pro1.exe)”命令,如图 C-6 所示。

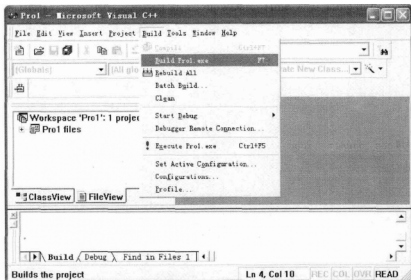


图 C-6

在选择了“Build pro1.exe (构建 pro1.exe)”命令后,系统按顺序编译项目中的各个文件。如果发现错误,将其错误信息显示在输出窗口中,并停止编译。修改其错误后,继续单击“Build pro1.exe”命令,则重新编译。第 1 个文件编译好后,再编译第 2 个文件,直到所有文件都编译好后,再进行连接。连接无错时,生成可执行文件 pro1.exe。

附录 C.5 运行项目可执行文件

选择了“Build (编译)→Execute pro1.exe (执行 pro1.exe)”命令,就运行项目可执行文件 pro1.exe,并将输出结果显示在弹出窗口中,如图 C-7 所示。



图 C-7

附录 C.6 关闭工作区

经过上述步骤，一个 C++ 多文件程序的编写工作就完成了。这时可以选择“File（文件）->Close Workspace（关闭工作区）”命令来关闭这个项目的工作区，结束该程序的操作。接着，就可以编辑和运行新程序了。若要退出 VC++ 环境，则执行“File（文件）->Exit（退出）”命令。

参 考 文 献

- [1] 陈维兴, 林小茶. C++面向对象程序设计教程(第3版). 北京: 清华大学出版社, 2009.
- [2] 陈维兴, 林小茶. C++面向对象程序设计. 北京: 中国铁道出版社, 2004.
- [3] Harvey M. Deitel, Paul James Deitel 著. 邱仲潘等译. C++大学教程(第二版). 北京: 电子工业出版社, 2002.
- [4] James P. Cohoon. Jack W. Davidson 著. 刘端挺等译. C++程序设计(第三版). 北京: 电子工业出版社, 2002.
- [5] Brian Overland 著. 董梁等译. C++语言命令详解(第二版). 北京: 电子工业出版社, 2002.
- [6] Richard C. Lee, William M. Tepfenhard 著. 麻志毅, 蒋严冰译. C++面向对象开发(原书第二版). 北京: 机械工业出版社, 2002.
- [7] 谭浩强. C++程序设计. 北京: 清华大学出版社, 2004.
- [8] 钱能. C++程序设计教程. 北京: 清华大学出版社, 2005.
- [9] 吴文虎. 程序设计基础[M]. 北京: 清华大学出版社, 2003.
- [10] 郑莉, 董瀚, 张端丰. C++语言程序设计(第3版). 北京: 清华大学出版社, 2003.
- [11] 罗建军, 朱丹军, 顾刚, 刘路放. C++程序设计教程(第2版). 北京: 高等教育出版社, 2007.